Extracting Data from a MOO

by
Curt Hill
Valley City State University
Curt_Hill@mail.vcsu.nodak.edu

Abstract

The usefulness of MOOs (Object Oriented MUDs, which are Multi User Domains) as an educational resource has been demonstrated in a variety of ways. For example, the ProgrammingLand Museum implements an Exploratorium-style museum metaphor to create a hyper-course in computer programming principles aimed at structuring the curriculum as a tour through a virtual museum. Student visitors are invited to participate in a self-paced exploration of the exhibit space, where they are introduced to the concepts of computer programming, are given demonstrations of these concepts in action, and are encouraged to manipulate the interactive exhibits as a way of experiencing the principles being taught. Although a MOO is a particular form of object oriented database that is easy to navigate for the student it is often difficult to obtain many kinds of global information from a MOO. Most MOO implementations have a variety of commands written in the script language, which answer some of the status and operational questions, but never all of them. Each such question requires a separate script program to answer. For example, finding all the exits labeled with the word 'next' without the alias 'n' is one such global question. Finding all the references to absolute object numbers in verbs or properties is another.

A system has been created where a MOO is systematically queried for information by a client program. This information is transformed into SQL statements and inserted into a relational database. The relational database can be queried much more easily than the corresponding MOO, for many important pieces of information. This system has three functional pieces. 1) A Java program communicates with the MOO in client mode using the telnet protocol. This program queries the MOO for status values and generates SQL statements. 2) A Microsoft SQL Server (or other SQL-capable database) processes these statements into a usable database. 3) A set of existing queries (with the inclusion of any new queries) is submitted to the SQL database which describe the workings of the MOO and the actions of students.

This paper discusses some of the problems involved in administering a MOO and the kinds of information that are tedious to extract. It then considers the developed system that eases this burden and gives several sets of example SQL statements that answer the questions posed earlier. Results from executions involving the ProgrammingLand MOO are given as examples.

The results of this work should be usable in any other environment using MOOs and the supporting programs are available for distribution.

Introduction: The Organization of a MOO

The use of MOOs for educational purposes is on the increase. They offer many advantages over web based instruction. [Hill and Slator, 1997] The World Wide Web Instructional Committee [WWWIC, 1998] is in the process of using and developing MOOs for the purpose of exposing students to highly interactive, virtual environments for educational purposes. A MOO is an immersive environment where activities and experiences will occur in a way difficult without costly laboratories and personnel.

The development of MOOs started with Multiple User Dungeons, an interactive multi-user dungeon and dragon style game. These evolved to also provide virtual social environments. [Keegan, 1997] A MOO is just an Object Oriented MUD. Pavel Curtis wrote the standard MOO software for a UNIX platform. [Curtis, 1997] Since that time it has been ported to several other platforms, such as Macintosh and Windows NT. [Unkel, 1997] The MOO consists of two pieces, the server software, which is written in C and a text database. At startup, the database is completely read into memory and then the server deals with clients using the telnet protocol. A basic database is called a core since it contains only the fundamental objects. There are several MOO cores available. The Lambda MOO core is the first and most common and is in use in the Geology Explorer MOO [Slator, Schwert, Saini-Eidukat, 1998] at NDSU. The encore High Wired core is specifically for educational MOOs [Haynes and Holmevik, 1997] and is the basis of ProgrammingLand on the VCSU campus.

The core itself is merely the starting point for any MOO. One of the strongest features of a MOO is that it is customized by a scripting language reminiscent of C. New objects can be created and endowed with whatever properties and methods (a method is called a verb in MOO terminology) are desired. Almost any aspect of the MOO is subject to modification by the proper manipulation of the properties and verbs of the MOO objects. A typical MOO core contains approximately one hundred objects, while a working MOO typically has several thousand objects.

The objects of a MOO have a few basic characteristics. Each object has an object number, which is prefixed by a # in most uses. This object number is unique for each object. All objects also have a name, a parent object, a location, an owner and several flags. The name is used to describe the object to any onlooker. The parent object is the super class. Every property and verb contained by the parent (and its parents) is also contained in this object. The object may have other properties and verbs added to it as well. The location is an object that contains this object. It is often a room, but may also be a

player. The owner of an object is a player who owns the object. The creator of the object is usually the owner. Players own themselves.

Objects can be categorized into four distinct classes, all derived from the basic MUD metaphor. Some objects are players (further subdivided into three classes), some are rooms, some are exits and all the rest are called things. A player uses the exits to navigate through the rooms and usually finds things scattered around the rooms. Most of these are objects that are descended from ancestral objects of that type. For example, object #7 is the Generic Exit and all exits have #7 as an immediate or distant ancestor. The others are: object #3 is the Generic Room, object #4 is the Generic Builder, object #5 is the Generic Thing and object #6 is the Generic Player.

A MOO is populated by three classes of players. The highest class is that of the wizard. The wizard has absolute power to create or destroy anything, consequently most MOOs have very few players trusted enough to be given wizard characters. Wizards typically own only core objects. Programmers have the ability to create new objects and to destroy anything that belongs to them. The bulk of a MOO is built by one or more programmers who must cooperate with one another in this process. Thus, a programmer or builder cannot add an exit to another programmer's room. They create the exit and then the other programmer must add it as an exit to their own room. (A wizard is not restricted by ownership in this way.) The last and lowest class is the player. Players explore the MOO, but cannot create rooms, exits or things.

A player uses a MOO by connecting to the MOO. This may be done with any program that supports the telnet protocol, though usually one of many MOO or MUD client programs are used. In the connection process, the player identifies their player name and password. Then they move through the MOO in a characteristic way. When they first enter a room, the description property is displayed. Any objects or players are mentioned as being present. Finally, visible exits are shown. The name of the exit is also the command to take that exit to the next room. While in the room the player may chat with any other player present in the room, since every room in a MOO is a chat room. If there are any other objects in the room, the player may interact with them in any way defined by the object. The player may pick up the object, should it not disallow such an action. However, in an educational MOO the other player and object interactions are usually more interesting.

The MOO and Education

The ProgrammingLand MOO and the Geology Explorer MOO are two examples of interesting but quite different educational approaches in the use of MOOs. The ProgrammingLand implements a Virtual Lecture with the

chief emphasis on instructional content. The Geology Explorer implements the Virtual Laboratory and its emphasis on goal directed experiments.

The ProgrammingLand MOO uses the metaphor of an Exploratorium. The student moves through a museum of programming, reading the exhibits in whichever direction the student chooses. There are several wings in the museum, each of which deal with a different programming language. Each room has one or two of paragraphs of text on a particular topic and directions to other rooms with related topics. A series of web pages could do this, but a MOO offers interactions not usually found in web pages. Code machines are one of the objects that populate the MOO. A programming student can observe a piece of code, have it explained on a line by line basis or trace through the execution of the code. This is the Virtual Lecture.

The Geology Explorer MOO uses a quite different metaphor. A student is given a goal of finding a particular mineral on a virtual planet. Each room in this MOO has the description of a particular terrain. Exits are the paths to other parts of the planet. Many rooms have several objects present which are various minerals and their descriptions show how they would look in a natural setting. The goal of recognizing the mineral not only requires knowing what it looks like in its native state, but also of testing that hypothesis in a more precise way. The student has the opportunity to start the exploration by purchasing the tools needed to test their minerals. Since the MOO knows the goal of the student's exploration, when they make a false step a tutor, which is an object in the MOO, can give them hints. These hints can include looking for a particular mineral without the needed tools or leaving the room where the mineral is present. This MOO gives the student an opportunity to have an experience that could be expensive or dangerous in the real world.

Administering a MOO

The administration of a MOO is not particularly difficult, but finding out certain details about the database can be. Administrators, who are usually the archwizard, are often forced to write special purpose MOO verbs to find out simple things. Many such verbs have made it into various cores, but this seems to be a bandage on a more serious problem. Consider the following problem: Most objects have one or more aliases to make their use easier. In a spatially oriented MOO the direction north will name an exit that proceeds in a northerly direction. Typing “north” every time turns out to be tedious, so the alias “n” is usually supplied. However, that alias must be applied to every exit, which cannot be done in any global fashion. The problem is finding all the exits that have “north” as a name or alias but do not have “n” as an alias or name. This requires the writing of a special verb that scans all exits and looks for this. It is not a large amount of work, but for all the similar questions, it is a new program or modification of an existing one. The typical strategy is to collect the complaints of players who explored the MOO and add the alias when someone complains.

This situation should remind the reader of the reasons for building database management systems in the first place. The answers to the important questions are in the data, but to find these answers requires myriad specially written programs, with incumbent costs. The reasonable solution then would be the construction of a database that contains the data of the MOO, but in a more accessible form. This is the germ of the idea on which this project is based. Collect the data from the MOO into a database that is easy to question for administrative purposes. A relational database seemed the obvious answer.

A Program to Extract Data from the MOO

The first problem is how to create the data for the database. There are two obvious approaches: scan the text version of the database without using the MOO server or use the server itself to scan the database. Although the MOO database is stored on disk in a textual form the parsing of that form is very difficult, every line in the disk file is either an integer or line of text. Some study of the problem showed that the first line displayed the version of the database format and the second the first unused object number. However, to determine the significance of each line of the database without documentation other than the server code was very daunting. Determining the characteristics of any object for a MOO wizard is quite straightforward: use the @show command. Therefore, an easier approach than interrogating the text database was the creation of a program that appeared to the MOO as just another client. Such a program would connect with wizard privileges and then systematically examine each object. It would then produce statements suitable as input to the database, mainly SQL Insert statements.

A word of explanation about the name may be in order. This program creates a Data Warehouse, that is a database that is a snapshot of an operational database. The process of Data Mining is processing a Data Warehous to find relationships that are not obvious. However, MOOs often refer to themselves as caves, dating back to the dungeon and dragons origins. The command to build a new room or an exit from one room to another is the @dig command. The MOOMiner program enters the cave, mines the raw data out and place it in relational database so that it can be refined into something usable. This is data mining indeed, but not to be confused with the more conventional use of the term in database field.

Java was chosen as the language to implement this program for two very good reasons. Java's platform independence is greater than any other language at this time, especially in reference to how it handles I/O. This is particularly important since such a program will do large amounts of I/O. Other MOO administrators could find the results useful, so a platform independent program would be more valuable than one confined to a single platform. The second reason is strongly related to the first. Such a program will need to implement the telnet protocol using a TCP/IP connection. Such a task is usually painful, but in Java, it is barely distinguishable from ordinary disk I/O.

The internal program details are not germane here, but a few details should be noted. The program consisted of 5 classes totaling less than 1500 lines of Java. The program starts with a menu that has four options: connect to the MOO; exit the program; set the host characteristics; and a description

of the program. The bulk of the program is an exercise in parsing the input that is returned by the various commands to display the objects. When the program runs to investigate ProgrammingLand it runs for about two hours and produces a file of SQL statements of size approximately 14 MB. The SQL server takes an additional four hours to process this file. All of these program times were running on a Pentium 233 with 64MB of RAM.

What *is* important here is a consideration of the basic database that the program provides. The entire output file is a collection of SQL statements. The first lines create the database and then the rest add records to the created schema. Every MOO scanned would have some basic items examined. However, there is the ability to look for items that may be specific to one particular MOO.

The central table of the database was the **objects** table which recorded the universal data about an object. The object_number is referenced by every other table in the database. All six-character fields in all the tables are object numbers. This object number is a pound sign (#) followed by a one to four digit number.

obj_name	object_number	parent	location	owner	is_player	is_programmer	is_wizard
varchar	char(6)	char(6)	char(6)	char(6)	bit	bit	bit

is_readable	is_writable	is_room	is_exit
bit	bit	bit	bit

The **verbdefs** table describes the verbs that are defined on any particular object. Recall that if an object defines a verb, all of its descendents may use it, however only the original contains the definition. When the server parses a command line, it categorizes the words into direct object, preposition and indirect object. These last three properties describe how the verb may use them should they occur in a command line.

object_number	verb_name	dir_obj	prep	indirobj
char(6)	varchar	char(10)	char(10)	char(10)

The **verbs** table captures all of the verb code, which are the objects methods. Each record represents one line of the verb program.

object_number	verb_name	line_number	the_line
char(6)	varchar(50)	integer	varchar(350)

The **verbrefs** table looks for absolute object numbers in the verb code. Such references are generally undesirable. Their presence complicates the updating of the core.

object_number	verb_name	refed_object	line_number
char(6)	varchar(50)	char(6)	integer

The **propdefs** describe what properties are defined on one object. Like the verbs, an object may define a property and all of its descendents will also have the property, but not define the property.

object_number	prop_name
char(6)	varchar(50)

The **property** table collects the values stored in any objects properties. The value will be present if the object defines the property or inherits the definition from any ancestor.

object_number	prop_name	seq	prop_value
char(6)	varchar(50)	integer identity	varchar(150)

The **proprefs** table parallels the verbrefs table. It is often the case that the value of a property is an object number, especially in object #0. This table captures any property that has as its value an object number.

object_number	prop_name	refed_object	seq
char(6)	varchar(50)	char(6)	integer identity

The **aliases** table captures a particular property that most objects have, that is alternative names for the object. One problem is that two objects may have the same name making referencing them difficult.

object_number	sequence	name
char(6)	integer identity	varchar(50)

The **dupverbs** table was developed near the end of the project. A MOO allows one object to have multiple verbs of the same name. Like C++ it distinguishes the verbs by parameter type. This table captures these duplicate verbs.

object_number	verb_name	seq
char(6)	varchar(50)	integer identity

The above tables are very general and could be used on any kind of MOO, but it is very important to be able to customize the program to the particular MOO. This program reads in a file that allows it to scan properties and create extra tables. Properties may have two kinds of values: scalars and lists. The property table can contain list values, but it will be difficult for a query to examine these. Therefore, a file is read at startup time that allows the creation of extra tables. Each table is specified by three items: the name of the table, the name of the property and the SQL variable description. Each such table has two supplied columns, one for the object number and another to guarantee uniqueness. Two such tables are next described that are useful for the ProgrammingLand MOO.

The **owns** table collects all the objects owned by a player. Only players who are programmers or wizards can actually own items. In ProgrammingLand the PGMR owns almost everything except the core, however, most of the LISP wing is owned by individual students. The owns

property is just a list of object numbers. This table is general enough that almost every MOO could use it.

object_number	seq	item
char(6)	integer identity	char(6)

The **history** table is unique to ProgrammingLand. Every student character possesses a property that lists every room the student has visited. The purpose of this was to be a diagnostic tool for the teacher and it is also used in certain active exits. An active exit allowed a student to pass if they had viewed the prerequisite material and warned them not to enter otherwise.

object_number	seq	item
char(6)	integer identity	char(6)

There is one additional way to customize the database. When all the objects have been examined, the program allows some additional SQL statements to be appended. The original motivation was to allow Views to be added after all the insertions were complete. Views substantially simplify the access to rooms and exits. Rooms and exits are objects like any other in a MOO, but they are very common and very important. Therefore it is convenient to add a view that shows all objects that are rooms and another those objects that are exits. This capability is handled merely by having a file of SQL statements that is appended to the regular file of statements. These may define views, indices or any other useful statement.

This project is not yet finished, in that other tables may be constructed as new questions arise or other features added. What questions have already been asked and answered by appropriate queries?

The SQL Queries

ProgrammingLand implements a museum of instruction on programming or a virtual lecture. An interesting question to ask is: how many exits occur from each room? The motivation for this question is the desire to avoid long hallways, where students meandering through the museum have few choices as to their next topics. Finding the average number of exits per room is quite easy, divide the number of rooms by the number of exits. The more difficult problem is finding the number of rooms with one exit or rooms with only two exits. A one-exit room is a dead end, usually the student must go back to room that was previously visited. A two-exit room is a hallway, the student can go forward or backward, but still has little choice. At the other extreme are rooms with many exits that may provide too many choices. The following query sorts the rooms by the number of exits:

```
select room.object_number, count(path.object_number)
from objects as room, objects as path, property as p
where room.is_room = 1 AND
    path.is_exit = 1 AND
    path.object_number = p.object_number AND
    p.prop_name = 'source' AND
    p.prop_value = room.object_number
group by room.object_number
order by count(path.object_number)
```

The solution to this query led to another related problem. ProgrammingLand has two distinct types of rooms. The normal room has some instructional text and then a number of exits are shown by the MOO. However, there are also signpost rooms which typically are the beginning of a lesson or group of lessons. The latter are usually menus of possible destinations. An example signpost room description follows:

> Flow of Control Statements
> Flow of control statements allow programs to execute statements in some fashion other than one right after another. Programs with only sequential flow are unable to respond to unusual situations, do anything more than once or many other interesting things.
> There are several kinds of statements that alter the normal flow of control, each of which have their own exhibits. Consider one of the following:
> a) Decision statement. Choose one of several alternatives. The most common is the if statement.
> b) Looping statements. There are three statements that provide repetitive execution, with several variations.
> c) Function calls or method invocation
> or

x) Return to the C++ foyer
You see smallif here.
Obvious exits: [exit] to C++ foyer, [decision] to Decision Statements, [loop] to Looping Statements, [function] to Functions

When a player enters a room, the MOO server shows the room name first and then the description, the contents and then possible exits. The name of the room is the first line. The next dozen lines comprize the description. This particular room has something in it, a code machine named smallif. If there were other students present they would also be shown. The final two lines are the visible exits. The name of the exit is in brackets and the name of the room that it leads to follows. Therefore a student can type *loop* and move to the beginning of the lesson on loops in C++. Any object may have a number of different aliases, the loop exit has an alias of *b*. Although it is not obvious from what is shown, the exit also has aliases of *loops* and *looping*. Thus the student may type any of these four and enter the looping room.

In most rooms those final several lines are the main indication as to where the student can go next. However, in a signpost room it is redundant and tends to make the display longer than needed. The MOO server will suppress the display of obvious exits if the room has a property called tell_exits set to zero. Therefore the following query was created to find all those rooms that had exits with an aliases of a, b, c and tell_exits of 1.

```
select r.obj_name, r.object_number, e1.obj_name, e2.obj_name,
  e3.obj_name, p.prop_value
from rooms r, exits e1, aliases as a1, exits e2, aliases as a2, exits e3,
  aliases as a3, property as p
where e1.source = r.object_number And
    a1.name='a' And e1.object_number = a1.object_number And
    e2.source = r.object_number And
    a2.name='b' And e2.object_number = a2.object_number And
    e3.source = r.object_number And
    a3.name='c' And e3.object_number = a3.object_number And
    p.prop_name='tell_exits' And p.object_number=r.object_number And
    p.prop_value=1
```

This particular query is usually only done once, since the rooms that it finds are usually changed to enhance the exploration of the MOO. The LambdaMOO script that would find all such rooms would be substantially more lines of code and work.

The students in this MOO have a property that contains all the room numbers that they have visited. The MOO program that displayed that was

relatively simple but substantially more complicated than the following SQL query:

```
select distinct st.obj_name, rooms.obj_name
from objects as st, history, objects as rooms
where st.object_number = history.object_number   AND
    history.item = rooms.object_number        AND
    history.item <> '#0'
order by st.obj_name
```

In the above what is shown is the player's name and the room's name. The list of items that have been visited usually starts with a zero, hence the exclusion of object zero.

The following query looks for objects that are not players, rooms, or exits that happen to be located in a room that has a different owner than the item itself. The situation that prompted this was students picking up instructional objects and moving them to other locations. This also finds objects that are owned by someone other than the player carrying him or her at the time.

```
select o1.obj_name, o1.owner, o2.obj_name, o2.owner
from objects as o1, objects as o2
where o1.location=o2.object_number   AND
    o1.owner<>o2.owner            AND
    not(o1.owner in ('#2','#3','#12','#13', '#121')) AND
    o1.is_player <> 1
```

The problem mentioned earlier in the paper, an exit labeled "next" that did not have an alias of "n" is shown next. The only modification needed for most MOOs is to change the 'next' in the third line to 'north':

```
select distinct o.object_number, obj_name, p.prop_value
from objects as o, aliases as a, property as p
where o.is_exit = 1 AND o.obj_name = 'next' AND
    a.object_number = o.object_number AND
    p.object_number = a.object_number AND p.prop_name='source' AND
    not a.object_number in (select object_number
                from aliases
                where name = 'n'
```

The last query to be considered, provides a list of all the players that own property. In ProgrammingLand four objects own the bulk of the MOO, the author's wizard character (#2); the authors builder (#121); hacker(#57) who is

a core item and not a real player; and the author's most common co-author (#135). These four are the normal owners, all the others are found with this query:

```
select owns.object_number, ow.obj_name, item, ob.obj_name
 from owns, objects as ob, objects as ow
 where owns.object_number != '#2' AND
      owns.object_number != '#121' AND
      owns.object_number != '#57' AND
      owns.object_number != '#135' AND
      owns.object_number = ow.object_number AND
      owns.object_number != item AND
      owns.item = ob.object_number
```

The above list of queries cannot be the exhaustive list. As the questions emerge new queries will be constructed. Appendix 1 also gives some queries that determine that the SQL statements produced by the program were inconsistent.

Future Work

There is much to do make this project most usable. The following list considers some improvements that will be considered.

1. The parsing of the output of the MOO is not based upon any special objects or characteristics of the MOO. Thus the examining program is subject to interruption. If a player enters the room the program is occupying the entrance message will be mixed with the output of the MOO, giving incorrect results. Furthermore, the parsing of the output is based upon about twenty messages that the MOO gives. Some of these messages are characteristic of the server, but most are characteristic of the MOO core. At the present they are all constant strings, but should be rendered in an external file to ease the customization to other cores. However, the relational database is the best tool to find where the properties that store the messages.

2. Many other standard queries need to be developed to gain more insight into the behavior of the students in the MOO. This may entail keeping more information in the student character as well as the generation of new SQL statement.

3. ProgrammingLand implements the virtual lecture. The normal use is for students to browse the exhibits in their own order as their curiosity demands. This is the desired mode for the initial visit to a room, but what about the use of the MOO as a reference tool. When the student has read something once and now wants to return quickly to check a detail, the MOO is ill equipped to supply a quick path to the item. One good solution to that is to add an extra property to the room, which is a list of keywords or phrases that the room considers. The relational database then becomes the tool to start the construction of an index or table of contents. Browsing the index sends the student immediately to the exhibit of interest. Such a property has not been implemented since there was no mechanism to exploit it in the MOO.

4. The MOOMiner program currently creates a rather large file of SQL statements that are later processed by the RDBMS. The experience with Microsoft's SQL Server is that finding errors in the SQL statements is a non-trivial problem. Even finding which statement was in error is not trivial. Furthermore, the extraction process is time consuming and build of the database is time consuming. A better approach is to have the program send the statements to the RDBMS online rather than through a file. The error tracking would be much better, since the Java program would know the statement that caused the error. The performance should also be better with the MOO server, SQL server and MOOMiner operating concurrently, usually on separate machines. This task was not handled to keep the project to a manageable size.

Project Evaluation

This project has been beneficial in a number of ways. The author has gained considerable understanding in many unrelated areas. These include the syntax and semantics of SQL, the vagaries of Microsoft's SQL Server, certain features of the Java language and even features of the MOO. For example the tell_exit property is not mentioned in the commonly used documentation of LambdaMOO. It was discovered by the author while looking for something else in query results. The potential of this project to make the MOO easier to use and administrate has only partially been illuminated. It is the author's desire that the project will be useful elsewhere as well.

Conclusion

This paper has discussed some of the problems of administering a MOO. These problems motivated the creation of a system to extract object information from the MOO via a Java application named MOOMiner. This program systematically examines each object in the MOO and converts the derived information into SQL statements. These statements are processed by a RDBMS, in this case Microsoft's SQL Server into a database that captures all the information on the MOO. This database is used to answer the questions posed by the MOO administrator. A number of predefined queries were discussed, but any other query can be made using normal SQL.

References

Curtis, Pavel (1997). The LambdaMOO Programmer's Manual. ftp://ftp.lambda.moo.mud.org /pub/MOO/ProgammersManual.html

Haynes, Cynthia , and Jan Rune Holmevik, eds (1997), High Wired: On the Design, Use and Theory of Educational MOOs. University of Michigan.

Hill, C. and Slator, B.M. (1998). Virtual lecture, virtual laboratory, or virtual lesson. Proceedings of the Small College Computing Symposium (SCCS98). Fargo-Moorhead, April. pp. 159-173

Keegan, Martin. (1997) A Classification of MUDs. Journal of MUD Research, volume 2, number 2 (July 1997). http://journal.tinymush.org/~jornr/v2n2/keegan.html

Slator, B.M., Schwert, D.P., Saini-Eidukat, B., and others. (1998). Planet Oit: a virtual environment and educational role-playing game to teach the geosciences. Proceedings of the Small College Computing Symposium, Fargo-Moorhead, April, pp. 378-392.

Unkel, Christopher (1997). WinMOO. http://www-personal.engin.umich.edu/~cunkel

WWWIC: Juell, P., McClean, P., Schwert, D., Saini-Eidukat, B., Slator, B., White, A. http://www.ndsu.nodak.edu/wwwic

Appendix A

The following queries evaluate the consistency of the extracted data:

```
select obj_name, is_player, is_room, is_exit
from objects
where (is_player = 1 and (is_room = 1 or is_exit=1)) OR (is_room = 1 and
  is_exit=1)


select kid.obj_name, kid.object_number, kid.is_room, kid.is_exit,
  kid.is_player, par.obj_name, par.object_number,  par.is_room,
  par.is_exit, par.is_player
from objects as kid, objects as par
where kid.parent = par.object_number AND (kid.is_player != par.is_player
  OR kid.is_room != par.is_room OR kid.is_exit != par.is_exit)
```

The first Select determines that no object is two of the following: room, player or exit. The player determination is just the presence or absence of a flag on the object. However, the MOO determines that an item is a room because it is descended from a room. This is difficult for the program since the descendent objects are not always in numerical order; e.g. object #50 may be a descendent of #287. Therefore the program looks as properties. If an object has properties characteristic of a room then the program concludes that it is a room. The first Select verifies that this conclusion was valid. All runs so far have been consistent.

The second Select attempts to verify the same thing in a different way. It compares all pairs where one item is the parent of the other and validates that both of the pairs are the same kind of thing. That is it should find any room that has an exit for a parent. This typically shows those items that are the beginning of the room or exit hierarchy. Thus the first runs have been consistent. The further queries are more interesting since they ask the questions that are hard to ask the MOO directly.

A third consistency issue is whether the various object numbers that are cited actually exist. Some such questions are caught by the database's referential integrity checking. However, checking that each parent object is an actual object cannot be done because parents may have larger or smaller object numbers than their descendents.