AMOS II Tutorial

Gustav Fahl and Tore Risch
Uppsala Database Laboratory
Department of Information Technology
Uppsala University
Sweden
Tore.Risch@it.uu.se
August 20, 2008 (revised 2011-01-30)

This is an introduction and tutorial to the AMOS II object-relational database system. AMOS II is a descendant of AMOS, which has its roots in a main memory version of Hewlett Packard's DBMS IRIS [1]. The entire database is stored in main memory and is saved on disk only through explicit commands. AMOS II can be used a single user database, a multi-user server, or as a collection of interacting AMOS II multi-database peers. In this tutorial only the single user version is described. AMOS II is an example of an object-relational peer DBMS having a declarative query language, AMOSQL. For a complete description of AMOS II's functionality, see [3][2].

1. Introduction

To be able to run the examples in the tutorial you need download Amos II zip-file to a directory (called the *Amos directory*). The unzip will create a subfolders *bin, demo, doc,* etc. The file *bin/amos2.exe* is the executable program and *bin/amos2.dmp* is a database that contains only the data needed by the AMOS II system itself. All the commands that are used in the tutorial are in the file *demo/tutorial.amosql*. In order to try out the queries at the end of the tutorial you also need to load the file *demo/wcdata.amosql*.

AMOS II is started from the OS command window by making the *Amos bin directory* the current directory and then issuing the OS command:

```
amos2
```

You will then enter the *AMOS II top loop* where you have the prompt:

```
Amos 1>
```

The number '1' in the prompt, the *generation number,* is used for the *rollback* command explained later. In the AMOS II top loop you can issue any AmosQL command to be described below.

To exit the system issue the command 'quit':

```
Amos 2> quit;
```

You are assumed to have general knowledge about databases and Entity Relationship modeling.

1.1 Styles used in the examples

This tutorial uses the following style for the AMOSQL commands you should enter to the AMOS II top loop during the tutorial:

```
Amos 4> create function name(Country)->Charstring as stored;
```

AMOSQL commands that are not to be entered but only to serve as examples look like this:

```
create type Tournament properties (year Integer, host Country);
```

Thus the AMOSQL commands that are supposed to be executed during the tutorial have the AMOS II prompt with generation number first.

2. An example

As a running example throughout the overview we use a database to manage information about World Cup tournaments in soccer. We start with an extended Entity Relationship schema (EER schema) that describes the information to be represented in the database. The steps how to implement the database in AMOS II are then gradually introduced as the database is built. The ER schema is shown in Figure 1. In Section 2.1 the corresponding database description is made in natural language.

The building blocks of the EER schema are represented by the following corresponding building blocks in the Amos II data model:

1. *Entity types* are represented as *types* in Amos II.
2. *Relationships* are represented as *functions*. The arrows in the EER diagram indicate the logical direction of the Amos II functions representing the relationships (from argument to result).
3. *Attributes* are represented as *functions* too.
4. *Cardinality constraints* are specified on functions by a special syntactic construct (Bag of, key).

2.1 Database description

World Cup tournaments in soccer are arranged every fourth year. Each tournament is arranged by some country. In each tournament a number of teams participate and they represent various countries. Each tournament contains a number of games (matches) each played between two teams. The team that made most goals wins. Each game is refereed by a referee and a certain number of spectators come to look at it. Each team contains a set of players. For each game played by a team of players a subset of these players are chosen to play in this particular match. A player can make a number of goals and a number of own goals in a certain game.

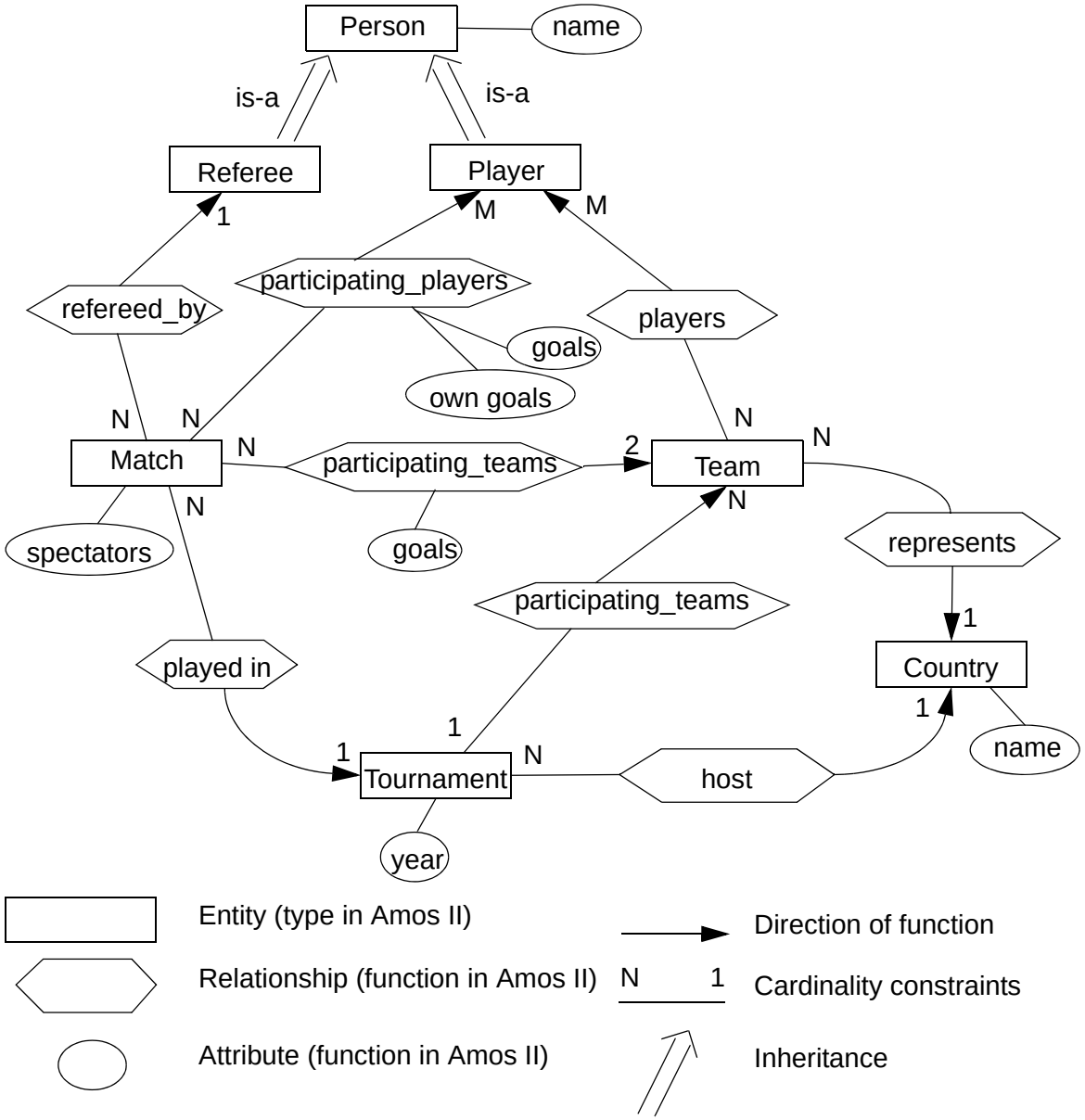

Figure 1 Extended ER schema describing the information in the World Cup database.

3. Practical advises

3.1 Rollback

The database can be restored to an earlier state by the command:

rollback *generation number*;

For example, after the following command all changes from command 4 and forward are undone:

rollback 4;

3.2 Reading commands from a file

AMOSQL commands can be read from a file and executed by the AmosQL command

< '*file name*';

For example, the AMOSQL commands for creating the tutorial database are stored in the file *tutorial.amosql*. To read an execute all commands in the file *demo/tutorial.amosql* when the current folder is the AmosII *bin* directory, execute the command:

Amos 1> < '../demo/tutorial.amosql';

Notice that in order to learn AMOSQL you are recommended *not* load the entire script, but to copy one command at the time from the file *demo/tutorial.amosql*. Use *cut and paste* for this.

3.3 Saving a database

A database can be saved in the file named *file name* by the command:

save '*file name* ';

To create the tutorial database and save it in the file *wc.dmp* do the following:

```
amos2
Amos 1> < '../demo/tutorial.amosql';
Amos 2> save 'wc.dmp';
Amos 1> quit;
```

To start AMOS II with the saved database, give the following command to the Windows command window:

amos2 *file name*

For example:

amos2 wc.dmp

4. Types, functions, and objects

The three basic concepts in AMOS II are *types*, *functions*, and *objects*. For those who are used to the terminology used with object oriented programming languages these concepts approximately correspond to *classes*, *methods*, and *instances.*

Types are used for classifying objects. Each object is an instance of a type. All properties of an object as well as relationships between objects are represented by functions.

4.1 Types

Entity types in an ER schema corresponds to *types* in AMOS II. We create types for the entity types *Country* and *Tournament* (see Figure 1) with the following AMOSQL commands:

```
AMOS 1>create type Tournament;
AMOS 2>create type Country;
```

Notice that names of types are case insensitive. Internally the system always stores the names of types in upper-case. However, by convention we always capitalize types in this tutorial.

4.2 Functions

The properties of objects and the relationships between objects are modeled as *functions*. In our example the entity type *Tournament* has one attribute *year*. We model this as a function *year* that has a *Tournament* as its argument and returns an integer:

```
AMOS 3>create function year(Tournament)->Integer as stored;
```

As for type names, function names are case insensitive in Amos II and they are internally upper-cased. In this tutorial we always write function names in lower-case.

In the same way the attribute *name* of the entity type *Country* becomes a function from countries to strings:

```
AMOS 4>create function name(Country)->Charstring as stored;
```

Between the entity types *Tournament* and *Country* there is the relationship *host*. This is modeled by a function from tournaments to countries:

```
AMOS 5>create function host(Tournament)->Country as stored;
```

The above functions are defined as *stored functions* (‘*... as stored*’). This means that the relationship between the argument and the result will be explicitly stored in the database, analogous to a table in a relational database. There are four kinds of functions in AMOS II. In addition to stored functions there are *derived functions*, *foreign functions*, and *stored procedures*. Derived functions are introduced in Section 7.1. Foreign functions are defined in a conventional programming language, usually Java. Foreign functions are not discussed in this tutorial. Stored procedures are functions having side effects, such as updating the database. Stored procedures are not covered by this tutorial.

Types in AMOS II can be *built in* or *user defined*. Examples of built in types are *Integer*, *Charstring*, and *Real*.

User defined functions are defined with the *create type* statement, e.g.:

```
create type Person;
```

Stored functions for a user defined type can be created together with the type definition. The commands 1-5 are equivalent to these commands:

```
create type Country properties (name Charstring);
create type Tournament properties (year Integer, host Country);
```

4.3 Objects

We are now ready to start populating the database by creating our first objects. The following command creates three countries, i.e. three instances of the type *country*:

```
AMOS 6>create Country instances :ita, :bra, :esp;
```

:ita, *:bra* and *:esp* are *environment variables* that are bound to the objects created. It is possible to refer to the objects through these variables only during the current transaction. They are *not* stored in the database! In the examples below environment variables are used to identify cross referenced objects in the example database.

For the type *Country* there is a function *name* defined. The following commands assign names to the three countries, i.e. defines the results of applying the function *name* on the three just created instances

of type *country*:

```
AMOS 7>set name(:ita)='Italy';
AMOS 8>set name(:bra)='Brazil';
AMOS 9>set name(:esp)='Spain';
```

In the same way we create three instances of type *Tournament* and define what values the functions *year* and *host* return when applied on the objects of type *Tournament*:

```
AMOS 10>create Tournament instances :t2, :t12, :t14;
AMOS 11>set year(:t2)=1934;
AMOS 12>set host(:t2)=:ita;
AMOS 13>set year(:t12)=1982;
AMOS 14>set host(:t12)=:esp;
AMOS 15>set year(:t14)=1990;
AMOS 16>set host(:t14)=:ita;
```

The results of the functions can also be defined directly when the objects are created. The commands 6-16 are equivalent with:

```
create Country(name) instances :ita ('Italy'),
    :bra ('Brazil'),
    :spa ('Spain');
create Tournament(year, host) instances :t2 (1934, :ita),
    :t12 (1982, :esp),
    :t14 (1990, :ita);
```

5. Query Language

Similarly to SQL, AMOSQL is a combination of a DDL (Data Definition Language) and a DML (Data Manipulation Language). The query part of the language is similar to SQL. The following command answers the query 'What country arranged the World Cup 1982?':

```
AMOS 17>select x
            from Charstring x, Country c, Tournament t
            where x=name(c)
              and c=host(t)
              and year(t)=1982;
```

An AMOSQL expression can be entered in one or several lines. The commands in AMOSQL are always terminated with semicolon (;).

Notice that one difference to SQL is that the FROM clause contains type declarations, not table names as in SQL.

The above query does not utilize the possibilities in AMOSQL to compose functions. The following command is equivalent to the previous one:

```
AMOS 18>select name(host(t))
            from Tournament t
            where year(t)=1982;
```

The commands 17 and 18 actually answer the query 'What *name* did the country have that arranged the World Cup 1982?'. If you really want to retrieve the country *object*, not its name, the command would be:

```
AMOS 19>select host(t)
        from Tournament t
        where year(t)=1982;
```

Notice here how the system returns *object identifiers* (OIDs) for the country objects rather than their names.

The following command answers the query ‘What years did Italy arrange the World Cup?’:

```
AMOS 20>select year(t)
        from Tournament t
        where name(host(t))=’Italy’;
```

As the final query example the following command answers the query ‘What years were the World Cup arranged and who were its hosts?’:

```
AMOS 21>select year(t), name(host(t))
        from Tournament t;
```

6. More about types

6.1 Type Hierarchy

The types in AMOS II are arranged in a *type hierarchy.* If a type A is defined as a subtype of a type B then the type A *inherits* all the functions defined on type B.

In our example the entity types *Referee* and *Player* are subtypes of the entity type *Person* that has an attribute *name*.

```
AMOS 22>create type Person;
AMOS 23>create type Referee under Person;
AMOS 24>create type Player under Person;
```

The attribute *name* is represented as a function *name* from persons to strings:

```
AMOS 25>create function name(Person)->Charstring as stored;
```

The function *name* is now defined on the types *Person*, *Referee*, and *Player* since *Referee* and *Player* are subtypes of *Person*.

If a type A is defined as a subtype of a type B it means that all instances of type A automatically become instances type B as well. To illustrate this, create the following instances of type *Person*, *Referee*, and *Player*:

```
AMOS 26>create Person (name) instances (’Pavel Smerdjakov’);
AMOS 27>create Referee (name) instances :r1 (’Abraham Klein’);
AMOS 28>create Player (name) instances :p1 (’Falcao’),
            :p2 (’Socrates’),
            :p3 (’Paolo Rossi’);
```

Then ask the following query and contemplate the answer:

```
AMOS 29>select name(p)
        from Person p;
```

We now create the remaining types from Figure 1:

```
AMOS 30>create type Team;
AMOS 31>create type Match;
```

7. More about functions

7.1 Derived functions

A function can be defined in term of other functions as a query. Such a function is called a *derived function*. For example, if we often need to find out the name of the host country for a given year, we can define a derived function *host_name* for this:

```
AMOS 32>create function host_name(Integer y)->Charstring hn as
                select name(host(t))
                from Tournament t
                where year(t)=y;
AMOS 33>select host_name(1982);
AMOS 34>select host_name(1990);
```

When a query consists of a single function call, as the queries 33 and 34, the *select* can be omitted. The queries 33 and 34 can thus also be written as:

```
host_name(1982);
host_name(1990);
```

7.2 Inverse function definitions

Consider the information that ‘countries have names’ in the EER schema of Figure 1 represented as an attribute *name* of the entity type *Country.* We represent this as in command 4, i.e. as a function from countries to names:

```
create function name(Country)->Charstring as stored;
```

The database is then populated with the commands:

```
set name(:ita)=’Italy’;
set name(:bra)=’Brazil’;
set name(:esp)=’Spain’;
```

To get the name of a certain country (here the country object *:ita*) one makes the query:

```
name(:ita);
```

Sometimes one might also be interested in the *inverse* function, i.e a function from names to countries:

```
create function country_named(Charstring cn)->Country c
   as select c from Country c where name(c)=cn;
```

Alternatively, since the variable *c* is declared also in the function result, the function *country_named* can be formulated without *from* as

```
create function country_named(Charstring cn)->Country c
   as select c where name(c)=cn
```

The database can then alternatively be populated by these commands:

```
set country_named(“Italy’)=:ita;
set country_named(“Brazil’)=:bra;
```

```
set country_named(“Spain’)=:esp;
```

One may get the name of the country object *:ita* by the query:

```
select cn
from Charstring cn
where country_named(cn)=:ita;
```

In general, if one thinks that several functions are useful, there is always the possibility to define one of them as a (stored) base function and then define the others as derived from the base function as in our example.

7.3 More function definitions

We define some more functions from Figure 1:

```
AMOS 35>create function represents(Team)->Country as stored;
AMOS 36>create function refereed_by(Match)->Referee as stored;
AMOS 37>create function played_in(Match)->Tournament as stored;
AMOS 38>create function spectators(Match)->Integer as stored;
```

7.4 Bag valued functions and cardinality constraints

When a stored function is defined, AMOS II by default assumes that the function returns *one* value, while the same result can be returned for different values of the argument, i.e. *many-one* relationships. For example, for the function *spectators(Match)->Integer* (command 68) it holds that one match can have one number as the number of spectators while several games can have the same number of spectators. Such constraints on the cardinality of function results or arguments are called *cardinality constraints*.

We model the relationship *participating_teams* between the entity types *Team* and *Tournament* as a function from tournaments to teams:

```
create function participating_teams(Tournament to)-> Team tm as stored;
```

In a tournament usually more than one team is participating. Therefore *participating_teams* is an example of a function that should return a set of more than one value. However, the stored function definition above does not allow this. To allow *participating_teams* to return a set of values you can specify ‘*Bag of Team*’ as result type[1]:

```
create function participating_teams(Tournament to)->Bag of Team tm as stored;
```

When you define the value of a stored function as ‘*Bag of*’ the function will represent a *many-many* relationship.

However, the function *participating_teams* actually represents a *one-many* relationship. For a certain value of the input parameter several values are returned (there are several teams participating in a tournament) while a certain result cannot be returned for more than one value of the input parameter (a team can only participate in one tournament). Therefore the ‘*Bag of*’ declaration is not fully correct. The recommended way to represent such a one-many relationship is to instead define the inverse function to represent the relationship. In our case we can define a function *tournament* instead of

1. A *bag* is a set where duplicates are allowed. Queries in AmosQL (and SQL) return bags rather than sets.

participating_teams:

```
AMOS 39>create function tournament(Team) -> Tournament as stored;
```

The function *participating_teams* can then be defined as the inverse of *tournament*:

```
AMOS 40>create function participating_teams(Tournament t)->Team tm
            as select tm where tournament(tm) = t;
```

Cardinality constraints can also constrain the uniqueness of the result of a function. The keyword '*key*' indicates that a parameter of a stored function is unique thus defining a *one-one* relationship.

For example, the functions *year(Tournament)->Integer* and *name(Country)->Charstring* (defined with the commands 3 and 4) have the constraint that the result must be unique. This uniqueness can be specified by adding the keyword 'key' after the result parameter. To include these cardinality constraints (one-one relationships), the functions should actually have been defined as:

create function year(Tournament)->Integer key as stored;

create function name(Country)->Charstring key as stored;

This makes it impossible to have two tournaments the same year or to have two countries with the same name. The system will raise an error when the constraint is violated.

For bag valued functions the command *add* is used for adding values to the results of function. The syntax is the same as for the *set* command:

```
AMOS 41>create Team (represents) instances :ita82 (:ita), :bra82 (:bra);
AMOS 42>add participating_teams(:t12)=:ita82;
AMOS 43>add participating_teams(:t12)=:bra82;
AMOS 44>name(represents(participating_teams(:t12)));
```

Notice that a function such as *participating_teams* which is the inverse of a stored function (*tournament*) is updatable. Normally derived function are not updatable. An equivalent way to create the two above teams using the inverse function tournament instead would be:

create Team(represents,tournament) instances :ita82 (:ita,:t12), :bra82 (:bra,:t12);

In Amos II, when a function returns a bag of values, as the inner call to *participating_teams* in command 44, the applied function (*represents*) is applied *on each element* of the result from the set valued function (*participating_teams*). This is called 'Daplex semantics' because it was first invented in the functional query language Daplex, an ancestor of AmosQL.

To model the relationship *player*s between *Team* and *Player* (see Figure 1) we define a function with that name from *Team* to *Player*. The relationship is many-to-many, i.e. a player can play in several teams and a team has several players. The function definition therefore becomes:

```
AMOS 45>create function players(Team)->Bag of Player as stored;
AMOS 46>add players(:bra82)=:p1;
AMOS 47>add players(:bra82)=:p2;
AMOS 48>add players(:ita82)=:p3;
AMOS 49>name(players(:bra82));
```

7.5 Functions returning tuples

All functions so far have had a single result type. It is also possible to have functions that return tuples of values.

In command 25 the names of persons were defined by the function:

```
create function name(Person)->Charstring as stored;
```

Another possibility could have been to separate the first and family names by defining a function that returns a *tuple* of two strings:

```
create function name2(Person)-> (Charstring, Charstring) as stored;
```

A person can then be assigned a name with the command:

```
set name2(:p1)=('Pavel', 'Smerdjakov');
```

Functions for retrieving the first and family names, respectively, can then be defined as derived functions calling *name2*:

```
create function first_name(Person p)->Charstring f as
    select f
    from Charstring l
    where name2(p)=(f, l);
create function last_name(Person p)->Charstring l as
    select l
    from Charstring f
    where name2(p)=(f, l);
```

7.6 ... and so the final function definitions

Now only two relationships from Figure 1 remain to be represented as functions: *participating_players* between *Player* and *Match* and *participating_teams* between *Team* and *Match*.

Let's start with the first relationship: 'Players are participants in games and makes a number goals and a number of own goals in them'.

The most interesting functions are 'What players participate in a certain match?', 'How many goals did a certain player make in a certain match?', and 'How many own goals did a certain player make in a certain match?':

```
create function participating_players(Match m)->Player p as ...
create function goals(Player p, Match m)->Integer g as ...
create function own_goals(Player p, Match m)->Integer og as ...
```

It is not recommended to define these functions as stored ones as it would create redundant data[1]. We rather store the information about the participation of players in games in a function *player_participation* and define the three functions above as derived:

```
AMOS 50>create function player_participations(Match)->
                        Bag of (Player, Integer goals, Integer own_goals) as stored;[2]
AMOS 51>create function participating_players(Match m)->Player p as
```

1. For example the fact that a player participated in a certain match would have been stored in three different places.
2. No variable names need to be specified after the types in the definitions of stored functions. However, for clarity we still specify the variable names *goals* and *own_goals*.

```
            select p
            from Integer g, Integer og
            where player_participations(m)=(p, g, og);
AMOS 52>create function goals(Player p, Match m)->Integer g as
            select g
            from Integer og
            where player_participations(m)=(p, g, og);
AMOS 53>create function own_goals(Player p, Match m)->Integer og as
            select og
            from Integer g
            where player_participations(m)=(p, g, og);
```

Notice here that the derived functions *participating_players*, *goals*, and *own_goals* return sets (bags) of result values and therefore should have the result type 'Bag of'. However, derived functions always return bags and the 'Bag of' declaration is therefore implicit for derived functions.

We store new data in the database and make some queries:

```
AMOS 54>create Match (refereed_by, played_in, spectators)
            instances :m1 (:r1, :t12, 44000);
AMOS 55>add player_participations(:m1)=(:p1, 1, 0);
AMOS 56>add player_participations(:m1)=(:p2, 1, 0);
AMOS 57>add player_participations(:m1)=(:p3, 3, 0);
AMOS 58>name(participating_players(:m1));
AMOS 59>select goals(p, :m1)
        from Player p
        where name(p)='Falcao';
```

The relationship 'Teams participate in games and make a number of goals in them' is implemented accordingly. The participation of teams in games is stored in the function *team_participation*. The two most interesting functions are 'What teams participate in a certain match?' and 'How many goals did a certain team make in a certain match?'. They are defined as derived functions:

```
AMOS 60>create function team_participations(Match)->Bag of (Team t, Integer goals) as stored;
AMOS 61>create function participating_teams(Match m)->Team t as
            select t
            from Integer g
            where team_participations(m)=(t, g);
AMOS 62>create function goals(Team t, Match m)->Integer g as
            select g
            where team_participations(m)=(t, g);
```

7.7 Aggregate functions

Aggregate functions compute aggregate values over bags:

sum(...) - returns the sum of a number of values.

count(...) - counts the number of values.

maxagg(...) - returns the highest value.

minagg(...) - returns the lowest value.

some(...) - equivalent with *count(...)>0* but much faster.

notany(...) - equivalent with *count(...)=0* but much faster.

Aggregate functions have somewhat different semantics compared to ‘ordinary’ AMOS II functions in that they are not applied on each element of a bag as regular functions, but on entire bags.

Consider the query ‘What players participated in the tournament *:t12*?’;

```
AMOS 63>players(participating_teams(:t12));
```

The function *participating_teams* is applied on the object *:t12*. With the data stored in the database at this point *participating_teams* returns two objects, *:bra82* and *:ita82*.

Then the function *players* is applied on *each one of the* two objects returned from the call *participating_teams*, *:bra82* and *:ita82*. *players(:bra82)* returns two objects, *:p1* and *:p2*. *players(:ita82)* returns a single object, *:p3*.

Thus the result of the complete AMOSQL query above is three objects bound to the environame variables *:p1*, *:p2*, and *:p3*.

Compare this with the query ‘How many teams participated in tournament *:t12*?’:

```
AMOS 64>count(participating_teams(:t12));
```

In this case the function *count* is applied once on the entire bag of objects from *participating_teams*, not once per result from *participating_teams*. The reason is that *count* is defined to take a *bag of objects* as argument:

```
create function count(Bag of Object x)->Integer r as ...
```

Nested queries can be used as arguments to aggregate functions. The following command answers the query ‘How many own goals have been made?’:

```
AMOS 65>sum(select own_goals(p, m) from Player p, Match m);
```

8. Miscellaneous

8.1 System functions

Several useful functions are predefined in AMOS II. See [3] for a complete list of these. Here we only give some examples of how they can be used:

```
AMOS 66>plus(7,4);
AMOS 67>7+4;
```

Most arithmetic functions can be written either in prefix or infix notation. Other examples of arithmetic functions are *minus (-)*, *times (*)*, *div (/)*, and *mod*.

```
AMOS 68>max(7, 4);
```

max and *min* returns the largest and the smallest of two objects, respectively. This is different from *maxagg* and *minagg* that return the largest and smallest object from *a bag of* objects. Notice that comparison is defined not only for numbers, but actually for any kind of object. For example:

```
max(‘a’,’b’);
```

returns the string *'b'*.

```
AMOS 69>select year(t)
        from Tournament t
        where year(t)>1945;
```

The comparison functions <, >, =, *!*=, >=, and <= are written in infix notation.

The comparison functions can be applied on numbers, strings, and any kind of objects. The result of comparing two arbitrary OIDs is, however, undefined (actually the internal OID numbers are then compared).

```
AMOS 70>typenamed('Player');
```

returns the type having the name *'Player'*.

```
AMOS 71>cardinality(typenamed('Player'));
```

returns the number of objects of the type named *Player.*

```
AMOS 72>allfunctions(typenamed('Player'));
```

returns all currently defined functions where some argument or result is of type *Player.*

9. More about the query language

9.1 Environment variables

It is possible to assign the value of a query to an *environment variable*.

```
AMOS 73>select c into :c
        from Country c, Tournament t
        where c=host(t)
           and year(t)=1982;
AMOS 74>name(:c);
```

One of the tuples retrieved that fulfill the selection criteria will be assigned. If the select statemnt returns many tuples it is undefined which one is assigned.

9.2 Examples of queries

In the file *'wcdata.amosql'* there are AMOSQL commands that populate the database. Read and execute these commands with these commands:

```
AMOS 75>< '../demo/wcdata.amosql';
```

Next follows some examples of somewhat more complicated queries:

1. 'In what tournaments did Sweden participate?'

```
AMOS 76>select year(t)
        from Tournament t
        where name(represents(participating_teams(t)))='Sweden';
```

2. 'How many games have been played by Sweden?'

```
AMOS 77>count(select m
                from Match m
                where name(represents(participating_teams(m)))='Sweden');
```

3. ’What is the total number of goals made by Sweden?’

```
AMOS 78>sum(select goals(t, m)
                  from Team t, Match m
                  where name(represents(t))=’Sweden’);
```

4. ’In what games were most goals made?’

A good start is to define a derived function *goals* that returns the total number of goals in a match. We make a first attempt:

```
AMOS 79>create function goals1(Match m)->Integer as
            select goals(t1, m) + goals(t2, m)
            from Team t1, Team t2;
AMOS 80>goals1(:m1);
```

In the match *:m1* Italy made three goals and Brazil two. The answer to the query thus ought to be ‘5’. As the function is currently define *t1* and *t2* can also be bound to the same team and therefore the erroneous answers ‘4’ and ‘6’ are returned. Thus we have to specify that *t1* and *t2* are different teams:

```
AMOS 81>create function goals2(Match m)->Integer as
            select goals(t1, m) + goals(t2, m)
            from Team t1, Team t2
            where t1 != t2;
AMOS 82>goals2(:m1);
```

The function is no longer returning illegal answers but instead it returns the correct answer twice. In one case *t1* is bound to Italy and *t2* to Brazil. In the other case *t2* is bound to Italy and *t1* to Brazil. To only use one of these cases we can use ‘<‘ rather than ‘*!=*’. The comparison function ‘<‘ can be used for comparing any kind of object and it thus can order any objects.

```
AMOS 83>create function goals(Match m)->Integer as
            select goals(t1, m) + goals(t2, m)
            from Team t1, Team t2
            where t1 < t2;
AMOS 84>goals(:m1);
```

We can now specify the original query:

```
AMOS 85>select m
        from Match m
        where goals(m)=maxagg(select goals(m1) from Match m1);
```

The answer is not very informative. We therefore define the function *matchinfo* that, given a certain match, returns a tuple of each participating team, how many goals the team made, and what year the match was played. Then we ask the query once more:

```
AMOS 86>create function matchinfo(Match m)->(Charstring n1,
                                  Integer g1, Charstring n2, Integer g2, Integer y) as
            select name(represents(t1)), goals(t1, m),
                        name(represents(t2)), goals(t2, m), year(played_in(m))
            from Team t1, Team t2
            where participating_teams(m)=t1
               and participating_teams(m)=t2
               and t1 < t2;
AMOS 87>select matchinfo(m)
        from Match m
```

```
        where goals(m)=maxagg(select goals(m1) from Match m1);
```

5. ’In what matches were more than ten goals made?’

```
AMOS 88>select matchinfo(m)
        from Match m
        where goals(m)>=10;
```

10. Exercises

If you typed something illegal or if there is some error you can recreate the saved correct database with the OS command:

```
amos2 wc.dmp
```

In Table 1 there is an overview of the types and functions that are defined.

1. Make a derived function *ref_name(Match)->Charstring* that, given a certain match, returns the name of the referee that refereed the match. Then use the derived function to answer query 2.
2. Which years were World Cup tournaments played?
3. A team is trained by a trainer. A trainer has a name and a salary. Create the types and functions needed to model this. Create a trainer object too. Store name and salary for the created trainer. Specify thereafter an AMOSQL command showing the name and salary for all trainers.
4. Who were the referees in the games where Tomas Brolin played?
5. Create a derived function *referees(Charstring)->Charstring* that, given the name of a certain player, returns the names of the referees that refereed games where that player participated. Then use the derived function to answer query 6.
6. How many times did Mexico arrange the World Cup?
7. What match had the most spectators? Assign the environment variable *:m* to the answer. Then use *:m* to retrieve the number of spectators and what teams that played in that match (use the function *matchinfo*).
8. What games did Sweden win?
9. How many goals did Kurt Hamrin make?
10. Create a function that returns the matches in which a given team was playing.
    Create another function computing the total number of goals for a given team.
    Use the function to compute the name of the country in the database making most goals.
    Why is the result repeated twice?
11. How many players are members of several teams?

Types	Functions
Tournament	year(Tournament) -> Integer
	host(Tournament) -> Country
	participating_teams(Tournament) -> Team
Country	name(Country) -> Charstring
Person	name(Person) -> Charstring
Referee (subtype of Person)	
Player (subtype of Person)	goals(Player, Match) -> Integer

Types	Functions
	own_goals(Player, Match) -> Integer
Team	represents(Team) -> Country
	players(Team) -> Player
	goals(Team, Match) -> Integer
Match	refereed_by(Match) -> Referee
	played_in(Match) -> Tournament
	spectators(Match) -> Integer
	player_participations(Match) -> (Player, Integer goals, Integer own_goals)
	participating_players(Match) -> Player
	team_participations(Match) -> (Team, Integer goals)
	participating_teams(Match m) -> Team
	goals(Match) -> Integer
	matchinfo(Match) -> (Charstring c1, Integer g1, Charstring c2, Integer g2,Integer y)

Table 1: Type and function definitions

11. Referenses

1 D.H.Fishman. et al.: *'Overview of the Iris DBMS', Object-Oriented Concepts, Databases and Applications*, W. Kim, F.H. Lochovsky (eds.), ACM Press, Addison-Wesley, 1989.

2 Tore Risch, Vanja Josifovski, Timour Katchaounov : *AMOS II Concepts*, **http://user.it.uu.se/~torer/publ/FuncMedPaper.pdf**

3 Staffan Flodin, Martin Hansson, Vanja Josifovski, Timour Katchaounov, Tore Risch, and Martin Sköld: *Amos II Release 10 User's Manual*, **http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html**