Object State Diagram in Secondo System

Victor Teixeira de Almeida

September 30, 2003

1 Introduction

This document is intended as a documentation of the state diagram of objects inside the Secondo system. This state diagram is needed because objects can have some persistent part handled by its own algebra. Every object in the Secondo system must have a memory part and may have a persistent part not handled by the Secondo system, but by the algebra from which the object belongs.

In this way, objects, when they exist in the Secondo system, can stay into two states: *opened* and *closed*. The figure 1 shows the state diagram for objects in the Secondo system. When an object is in *opened* state, it has the memory part already loaded into memory and the files/records needed for the persistent part (if used) are opened and ready for use. When an object is in *closed* state the files/records needed for the persistent part are closed and the memory part is freed.

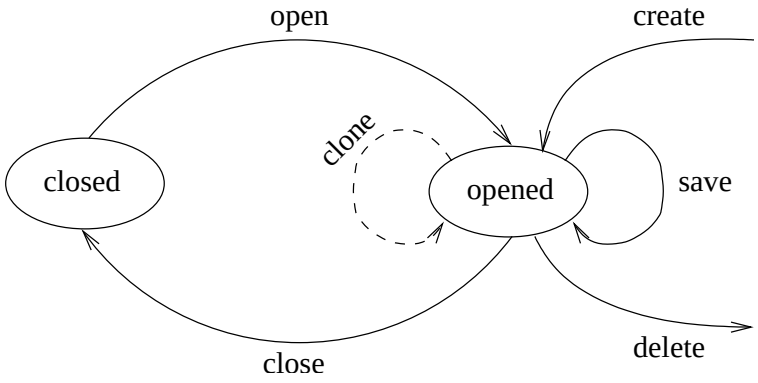

Figure 1: Object state diagram.

Every type constructor must implement these six transition functions, namely *create*, *delete*, *open*, *close*, *save*, and *clone*. The *create* transition creates an object in the system and allocates memory for its memory part and creates files/records (if needed) for its persistent part, in summary, it

looks like a constructor of the object. The *delete* transition is the contrary of the *create* one, it takes an opened object and deletes both memory and persistent parts. Like the *create* transition, the *delete* one looks like the destruction of the object. The *close* transition takes an opened object and closes it without deleting it, i.e., deletes the memory part and closes the persistent part. The *open* transition puts again an object into the *opened* state allocating the memory part of the object and reading the persistent part from file/records. The *save* transition is applied to an opened object and does not change the state of the object, but only saves the persistent part of it. Finally, the *clone* transition also takes an opened object and does not change it. This transition returns a newly created object (in opened state) which is a copy of the first one.

How the main commands for objects (*create*, *update*, *delete*, and *query*) use these transition functions will be presented in the next sections.

1.1 Create command

Creation of objects in the Secondo system only store necessary information in the catalog and do not really call the create function for storing space for the object in the system. The objects are created with the *defined* flag set to *false*, so the query processor knows that they were not initialized.

1.2 Delete command

This command deletes an object previously stored in the Secondo system. It needs then to call the *delete* transition function, but once all objects are in *closed* state, this command needs to open it first. If the object is not defined then this command do nothing. The example below shows the sequence of transition function calls for the delete command.

```
delete x

if( defined )
{
     open( x );
     delete( x );
}
```

1.3 Update command

The update command must be divided into three categories. The first one is when an object is being updated with a value calculated by the query processor. This value can be the result of a math computation, for example,

or the application of an operator, or even a series of operators. The example below shows the sequence of transition function calls for the this first kind of update command. In this kind of update command, the object x needs first to be deleted if it is defined. Then a new object is created by the query processor, the 0 is done, and the object x is saved and closed. The opening and closing of the objects (y) used in the right side of the 0 will be done by the query processor and will be explained in the *query* command.

```
update x := [y]

if( defined )
{
     open( x );
     delete( x );
}
save( x );
close( x );
```

The second type of an update command is when a real update of an object occur. The object, in this kind of update, is not being assigned a value, but updating itself. The example below shows the sequence of transition function calls for the this second kind of update command. This command can be the resulting of an operation like $inc(x)$, for example, which takes an integer value x and increments its value by 1. In this kind of operation, it is just needed to save the new value of the object x and close it. The object x must be necessarily created before this command.

```
x := [x]

save( x );
close( x );
```

The third and the last kind of update command is when an object is assigned directly a value of another object. The example below shows the sequence of transition function calls for the this third kind of update command. This case need to be handled differently because a clone of the object y in the right side of the 0 must be done before the real 0. This command begins like the first one deleting (if necessary) the object x. Then, the object y is opened, cloned, and assigned to the object x. The object x is then saved and both are closed.

```
x := y

if( defined )
{
     open( x );
     delete( x );
}
open( y );
x = clone( y );
save( x );
close( x );
close( y );
```

1.4 Query

In queries, the object creation, deletion, opening, and closing will be done by the query processor. The query processor constructs a query tree which looks like the one in figure 2. The circles represent internal nodes of the tree where the objects need to be created and then deleted to receive the result of operations. The rectangles represent leaf nodes of the tree where there can exist objects and constants. If the leaf node contains an object, the objece needs only to be opened and closed. If it contains a constant, the constant needs to be created and deleted.

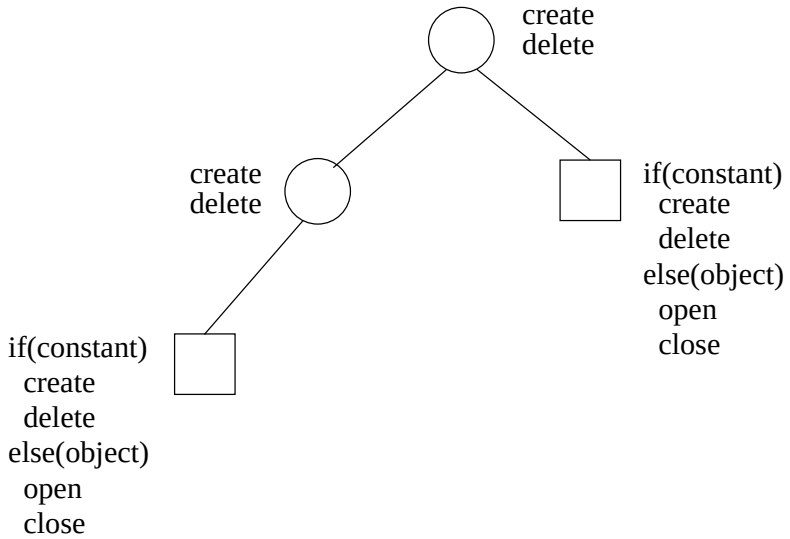

Figure 2: Object manipulation in the query tree.