Monikers in the Bonobo Component System

Miguel de Icaza
miguel@ximian.com

1 Introduction

We recently reimplemented and fully revamped the the moniker support in Bonobo. This work has opened a wide range of possibilities: from unifying the object naming space, to provide better integration in the system.

Note: on this document I have omitted exception environments handling for the sake of explaining the technology.

2 Monikers - a user perspective

Monikers are used to name objects, they effectively implement an object naming space. You can obtain monikers either because you created the moniker manually, or from a stringified representation of a moniker.

Here is a list of stringified monikers, and an interpretation of it:

`file:quake-scores.gnumeric`
This would be a moniker that represents the file quake-scores.gnumeric

`oafid:GNOME:Gnumeric:WorkbookFactory:1.0`
This moniker represents the Gnumeric Workbook factory object.

`oafid:GNOME:Gnumeric:WorkbookFactory:1.0:new:`
This moniker represents a Gnumeric Workbook instance. Notice that we are using the exact same OAFID as the example before, but there is a "new:" suffix at the end.

`file:/tmp/a.gz`
This represents the file in /tmp/a.gz

`file:/tmp/a.gz#gunzip`
This represents the decompressed stream of data from a.gz

`file:/tmp/a.gz#gunzip:streamcache`
This provides a cache on top of the decompressed stream of data for a.gz (the streamcache moniker is an in-proc component).

`http://www.gnome.org`
This one represents the GNOME web site.

`evolution:Mail/Inbox`
This represents the Evolution Mail/Inbox folder

`file:quake-scores.gnumeric!January`
This represents the January Sheet in the quake-scores.gnumeric workbook.

`file:quake-scores.gnumeric!January!Winner`
This represents the cell whose name is "Winner" in the January sheet in the quake-scores.gnumeric workbook.

`file:quake-scores.gnumeric!January!Winner!Style!Font`
This represents the Font interface of the Style attached to the Winner cell.

`file:quake-scores.gnumeric!January!Winner!Style!BackgroundColor`
This represents the background color for the cell.

`http://www.gnome.org/index.html!title`
This represents the title element in the HTML web page at www.gnome.org

`file:toyota.xml!cars/car/model/`
The "cars/car/model" is an XPath expression that for locating a specific node in the toyota.xml file.

`config:*/Session/Calendar`
This represents a PropertyBag for the GNOME Calendar using the Local configuration system and using the settings stored in the Session domain.

`oafid:Helix:Evolution:Wombat:1.0`
This represents the Evolution model server that

stores all the per-user information.

`queue:oafid:Helix:Evolution:Wombat`
This represents an interface that queues CORBA requests to the Evoution Wombat: Any calls issued will be queued: if the Wombat is busy or not accepting connection, all the CORBA method invocations will be queued without stopping the execution of the client code.

`http://www.gnome.org/index.html.gz#gunzip#html:title`
This will reutrn the title element of the compressed HTML file at http://www.gnome.org/index.html.gz

`ftp://ftp.gnome.org/gnome-core-1.0.tar.gz#utar/gnome-core-1.0/ChangeLog`
A reference to the ChangeLog file contained in the compressed gnome-core-1.0.tar.gz tar file at ftp://ftp.gnome.org

`desktop:Backgound`
The background object for the user's desktop.

`trashcan:`
The system trashcan.

`file:logo.png`
This represents the logo.png file.

`oafid:OAFIID:eog_viewer_factory:file:logo.png`
This specifies a specific image viewer to be used to display the file "logo.png", in this case the "EOG" program.

`file:logo.png!Zoom=2.0`
This represents the logo.png file in EOG at zoom level 2.0

`file:logo.png!Zoom=2.0,dither=max,notransparency`
The image logo.png is configured to be zoomed at 2.0 factor, to do maximum dithering and not use any transparency

Now, what you saw above are some examples of stringified representations of monikers. This means that they are not really monikers, it is the way a Moniker is represented in string form.

Monikers typically are created either by using a Bonobo API call that transforms the stringified representation into an object (which exports the IDL:Bonobo/Moniker:1.0 interface), like this:

```
moniker = bonobo_moniker_client_new_from_name
              (moniker_string);
```

Now, a moniker is only interesting because it can yield other objects when resolved. During the resolution process, you specify which interface you are intersted on the moniker to return. This is achieved by invoking the ::resolve method on the moniker and passing the repoid of the interface you desire, like this:

```
Bonobo::Unknown control;

control = moniker->resolve ("Bonobo/Control")
```

This would request the moniker to return an object that implements the IDL:Bonobo/Control:1.0 interface. This means that the object could be embedded as a regular Bonobo control in applications.

Maybe you do not want to get a control, but rather to resolve the moniker against a different interface, for instance a Bonobo::PropertyBag interface:

```
properties = moniker->resolve ("Bonobo/PropertyBag");
```

The resolution process might yield completely different objects.

The parsing and resolution process is all encapsulated into a single API call for your convenience: the bonobo_get_object function:

```
Bonobo::Unknown foo = bonobo_object_get
                          (char *moniker_string,
                           char *interface);
```

Now, as I said, the resolution process might yield very different objects depending on the interface being requested, for example:

```
x = bonobo_object_get ("http://www.gnome.org",
                       "Bonobo/Control")
y = bonobo_object_get ("http://www.gnome.org",
                       "Bonobo/Stream")
```

The "x" object might launch Mozilla which would in turn load www.gnome.org, and the returned object can be used as a Bonobo Control, and used in your application as a widget.

The "y" object on the other hand does not need all the power of Mozilla, we are only requesting the very simple Stream interface, so we might be able

to implement this with a lightweight HTTP implementation: maybe a wget-based bonobo server, or a libghttp server.

Note that even if the stringified versions of the monikers were the same (i.e., `http://www.gnome.org`) the resulting objects might differ wildely depending on the interface being requested.

3 The moniker parsing system

During parsing the moniker stringified, Bonobo will use the colon-terminated prefix as the toplevel moniker to be invoked for the resolution process.

For the prefix `file:` the file moniker will be used; For the prefix `oafid:`, the oafid moniker will be used; For the `queue:` prefix, the queue moniker will be used.

Once the moniker that handles a specific prefix has been activated, the moniker will be requested to parse the remaining of the string specification and return a valid moniker.

Each moniker typically will consume a number of bytes up to the point where its domain stops, will figure out what is the next moniker afterwards. Then it will activate the next moniker and pass the remaining of the moniker stringified version until the parsing is finished.

Each moniker is free to define its own mechanism for parsing, its special characters that are used to indicate the end of a moniker space, and the beginning of a new one (like the "#" and the "!" characters in some of the examples above). This flexibility is possible because each moniker gets to define its rules (and this is important, as we want to integrate with standards like http and file).

4 Monikers as an object naming scheme

As you can see, monikers are used to implement a naming system that can be used to reference and manipulate objects. As you might have noticed, the `::resolve` method on the moniker interface returns a `Bonobo::Unknown` interface. And by definition, the bonobo_get_object also returns a `Bonobo::Unknown`.

This means that the resulting object from the moniker resolution will always support `ref`, `unref` and `query_interface` methods.

The moniker object naming scheme is:

Extensible. A new entry point into the object naming space can be created and installed into the system. This is achieved by installing a new component.

Hierarchial

4.1 Creating monikers

Monikers are created typically by API calls into the Bonobo runtime or by your own classes that implement monikers.

4.2 Comparing the moniker name space with the Unix name space

Lets start simple. A moniker is a reference to an object. To actually use the object, you have to "resolve" the moniker. The term used in the literature is "binding the object".

The result of resolving the moniker is a Bonobo::Unknown object.

Think of a moniker as a pathname. And think of the binding process as the "open" system call on Unix.

The following is a list of Unix/moniker "mappings."

Object naming:
: path name ↔ moniker string representation.

Binding function:
: open(2) ↔ bonobo_get_object

Return value:
: kernel file descriptor ↔ Bonobo::Unknown CORBA reference

Binder:

Kernel VFS + each FS ↔ bonobo_get_object + Bonobo::Moniker

Persisting:

none ↔ Moniker::QI(Persist)

In the case of the file system, the kernel does the "resolution" of each path element by parsing one element of the file system, and the Virtual File System switch uses the current file system + mount points to resolve the ultimate file name.

4.3 File linking

Monikers were originally implemented as part of the Microsoft OLE2 compound document system. They can be used effectively by applications during drag and drop and cut and paste operations to pass objects that must be linked by other applications.

The source application would create a moniker for a given object that would fully identify it, and pass it trough a drag and drop operation or a cut and paste operation to the recipient application.

The recipient application then can resolve the moniker against the interface required (in the Bonobo case, Bonobo/Embeddable, or Bonobo/Control would be a common choice).

Applications do not need to store the entire contents of linked information, they can just store a stringified representation of the moniker, and resolve it again at load time.

4.4 Instance initialization

Monikers can be used to initialize objects, as a way of passing arguments to your object. This is coupled with the Bonobo/ItemContainer interface and the Item Moniker.

The Item Moniker is covered later.

4.5 Resolution of a moniker against an interface

A moniker can be resolved against different interfaces. The resulting object might be different depending on the interface that is being resolved. To illustrate this, here is an example, lets say we have the `http://www.helixcode.com` string representation of a moniker.

The string representation of the moniker can be resolved against the Bonobo/Control interface:

```
bonobo_get_object ("http://www.helixcode.com",
                   "Bonobo/Control");
```

This could return an embeddable Mozilla component that is suitable to be embedded into your application as a widget (because we are requesting the moniker to return a Bonobo/Control interface). If the interface is resolved against the Bonobo/Stream interface, maybe Mozilla is not required, and the process could use a smaller process that just provides Bonobo/Streams, say a CORBA-ified wget.

The logic for this lives on the `http:` moniker handler.

5 Core monikers

Bonobo ships with a number of moniker handlers: the file moniker, the item moniker, the oafid moniker and the new moniker.

5.1 The file moniker

The file moniker is used to reference files. For instance:

```
file:sales.gnumeric
```

The file moniker will scan its argument until it reaches the special characters '#' or '!' which indicate the end of the filename.

The file moniker will use the mime type associated with the file to find a component that will handle the

file. Once the object handler has been invoked, the Moniker will try to feed the file to the component first through quering the PersistFile interface, and if this is not supported, through the PersistStream interface.

5.2 The item moniker

The item moniker is typically triggered by the "!" string in the middle. The item moniker can be used to implement custom object naming, or argument handling.

The item moniker parses the text following '!' until the next '!' character, this is called the argument of the item moniker. During the resolution process, the item moniker will request from its parent the Bonobo/ItemContainer interface and will invoke the getObject on this interface with the argument.

For example, in a Gnumeric spreadsheet this allows programmers to reference sub-objects by name. For instance, Workbooks can locate Sheet objects; Sheets can locate range names, cell names, or cell references. This moniker would reference the sheet named 'Sales' in the workbook contained in the sales.gnumeric spreadsheet:

```
sheet = bonobo_get_object ("sales.gnumeric!Sales",
                           "Gnumeric/Sheet");
```

This other would reference the cell that has been named 'Total' inside the Sheet "Sales":

```
cell = bonobo_get_object
           ("sales.gnumeric!Sales!Total",
            "Gnumeric/Cell")
```

The way this works from the container perspective, is that the container will implement the getObject (string) method, and would respond to the getObject request.

Item monikers can also be used to perform instance initialization. The component that wants to support instance initialization needs to support the Bonobo/ItemContainer interface and implement a getObject method that would return the object properly initialized.

For example, lets consider an image viewer component that can be configured, like this:

```
image = bonobo_get_object
            ("file.jpg!convert_to_gray=on",
             "Bonobo/Control")
```

The above example would activate the EOG component because of the file.jpg match, and then invoke EOG's ItemContainer implementation with the argument "convert_to_gray=on". getObject should return an object (which would be itself) but it would modify the instance data to set the "convert_to_gray" flag to on. Like this:

```
Bonobo_Unknown
eog_item_container_get_object
    (BonoboObject *o, char *name)
{
    if (command_is (name, "convert_to_gray", &v))
        image_set_convert_to_gray (o, v);
     ...
    bonobo_object_ref (o);
    return bonobo_objet_corba_objref (o);
}
```

5.3 The oafiid moniker

The oafid moniker handles activation using the Object Activation Framework. This allows application programmers to activate objects by their OAF ID, like this:

```
gnumeric = bonobo_object_get
               ("oafiid:GNOME_Gnumeric_Workbook",
               iface)
```

5.4 The new moniker

The new moniker requests from its parent the "Bonobo/GenericFactory" interface and invokes the method create_instance in the interface.

Typically this moniker would be invoked like this:

```
bonobo_get_object ("oafid:RandomFactory:new:",
                   iface);
```

In the example above `RandomFactory` is the OAFID for the factory for a certain object. During the resolution process, the `new:` moniker would request its parent to resolve against the IDL:GNOME/ObjectFactory:1.0 interface (which is

the traditional factory interface in GNOME for creating new object instances) and then invoke the new_instance method on it.

Historically GNORBA (the old GNOME object activation system) and OAF (the new object activation system) implemented a special "hack" to do this same processing. Basically, the description files for the object activation system was overloaded, there were three types of activation mechanism defined:

1. activate object implementation from an executable.
2. activate object implementation from a shared library.
3. activate object implementation by launching another object, and querying the launched object for the ObjectFactory interface.

The `new:` moniker basically obviates the need for the last step in the activation system. With OAF, using the OAF approach proves to be more useful, as it is possible to query OAF for components that have certain attributes, and the attributes for a factory object are not as interesting as the attributes for the instances themselves. Despite this, the "new:" moniker can be used for performing the operation of instance initialization in more complex scenarios that go beyond the scope of activation provided by OAF.

6 Adding moniker handlers to the system

6.1 Ideal monikers

There are two moniker handlers that would be interesting to implement: the Configuration Moniker and the VFS moniker.

They both help the system overall, because the added simplicity of having a standard way of activating services in the system and given that the API to these services is CORBA-based, any programming language with CORBA/Bonobo support can make use of them without the need of a special language binding.

I am convinced that this helps make the system more self consistant internally.

6.2 The configuration coniker

The configuration moniker is invoked by using the `config:` prefix. The string afterwards is the configuration locator. The moniker should support being querried against the "Bonobo/Property" or "Bonobo/PropertyBag" depending on whether we are requesting a set of attributes, or a single attribute.

For example, retrieving the configuration information for a specific configuration property in Gnumeric would work like this:

```
Bonobo_Property auto_save;
CORBA_Any value;

auto_save = bonobo_get_object
                ("config:gnumeric/auto-save",
                 "/Bonobo/Property");
value = bonobo_property_get_value (auto_save,
                                   &ev);

if (value->tc->kind == CORBA_tk_bool)
    printf ("Value: %s\n",
            (CORBA_bool)value->_value ?
                        "true" : "false");
else
    printf ("Property is not boolean\n");
```

In the above example, we first use the bonobo_get_object routine to locate the configuration object through its moniker. The return value from the bonobo_get_object is of type Bonobo_Property which is the standard Bonobo way of manipulating properties.

This has two main advantages. First, by accessing the configuration engine through the moniker interface we have eliminated the need to define a C-specific API for the configuration management. The configuration could have been reached through any other programming language that supports CORBA.

The GNOME project has always tried to define APIs that could be easily wrapped and accessed from various languages (particularly, we have done this with the toolkit and recently with the CORBA bindings).

But even if we have taken special care of doing this, and there are continous efforts to wrap the latest and greatest APIs, widgets, and tools, the bindings typically lag a few weeks to monthsw behind the actual C API.

By moving towards CORBA, we only need to support CORBA in the various programming languages and we get access to any new APIs defined by it.

Second, any tools on the system that can manipulate a Bonobo::Property or ::PropertyBag (a GUI in a visual designer, or a configuration engine that persists/hidrates objects, or a browsing tool) can talk directly to the configuration engine all of a sudden, as we are using the same interface method across every language on the system.

The Bonobo::Property interface is pretty comprehensive, and should address most needs, the methods are as follows:

```
string   get_name ();
TypeCode get_type ();
any      get_value ();
void     set_value ();
any      get_default ();
string   get_doc_string ();
long     get_flags ();
```

Now, this interface as you can see does not specify an implementation for the actual backend. Given that this is just an interface, we do not care what the moniker will connect us to, we only care with the fact that we will be able to use the Property and PropertyBag interfaces.

6.2.1 Configuration transactions

Handling of transactional changes to the configuration system can be achieved by the use of the setValues interface in the PropertyBag. The implementation of the PropertyBag can either accept the values set, or it can do consistency checking of the values being set (for instance, to avoid the configuration to contradict itself, or store invalid values). If the values being set are invalid, an exception is thrown.

It would be also possible to hook up an arbitrary consistency checking component in the middle, by inserting the consistency checking in the middle of the stream, like this:

```
bonobo_get_object
    ("config:gnumeric/auto-save:gnumeric-consistency-check:",
     "Bonobo/Property");
```

Notice the `gnumeric-consistency-check:` moniker handler. This could just be a shared library consistency checking component if it needs to be.

6.2.2 Listening to changes

One of the requirements for a modern desktop is to be react globally when changes are made to global settings. For example, in the GNOME desktop when a theme is changed, a special protocol inside Gtk+ is used to notify all the applications that they should reload their theme configuration.

There are many other examples where applications need to keep track of the current setting. For example, when a preference is changed, we want the preference to take place right away, without us having to restart our running applications.

This is easily achieved by registering a Listener with the Bonobo/EventSource in the PropertyBag.

6.2.3 What about GConf?

GConf is a configuration management infrastructure that provides the following features:

1. A schema system for specifying the various configuration options, as well as their documentation and initial values (default values).
2. A way for the system administrator to override values in a system-wide fashion (this encompasses a network-wise setup if desired).
3. A change notification system: applications might be notified of changes to various values they might want to keep track of.

There are two drawbacks to GConf currently:

1. Although GConf provides pretty much everything that is required, but it is a C-based API that needs to be wrapped for every language that wants to support GConf.

2. GConf is limited in the kind of information that can be stored on its database. A BonoboProperty stores a CORBA_Any which can contain any of the simple CORBA types (strings, integers, floating points, booleans), structures, arrays and unions.

The actual engine and backend for GConf could become the configuration moniker handler, only the API would be replaced as well as the actual storage system to support the more complete CORBA_Any, and the ad-hoc CORBA interface can be replaced with a more powerful system.

6.2.4 Configuration management: Open Issues

Specifying the location for configuration The syntax for accessing the configuration has not been defined, but we can cook this up pretty easily.

Forcing the configuration data to be loaded from a specific location. Although the arguments to the moniker could be used to encode a specific location, for example:

```
config: /file.config!auto-save
```

It seems more natural to use the file moniker to provide this information, for example:

```
file: /file.config!config:auto-save
```

The config moniker can test for the presence of a parent, and if the parent exists, then it would request one of the Persist interfaces from it to load the actual configuration file, and provide access to it.

Transactional setting of values It might make sense to "batch" a number of changes done under a prefix to avoid listeners to a range of keys to reset themselves multiple times. Consider the case in which a command line tool makes various changes to the background properties, say the changes are done in this order:

```
background = bonobo_get_object
                ("config:desktop/background",
                 "PropertyBag");
bonobo_property_bag_set_values (background,
bonobo_property_list_new (
"gradient", "string", "true",
"color1",   "string"  "red",
"color2",   "string"  "blue",
&ev));
```

If the real configuration program for handling the background is running at that point, it will have registered to be notified of changes to all those values. The changes might be very expensive. For example the code migh react to every change and recompute the whole background image on each change.

An optimization would be to tag the beginning of the transaction and the end of it in the client code to allow listeners to get batched notification of changes:

```
background = bonobo_get_object
                ("config:desktop/background",
                 iface);
bonobo_property_bag_batch_push (background);
bonobo_property_set (background, "gradient", "true");
bonobo_property_set (background, "color1", "red");
bonobo_property_set (background, "color2", "blue");
bonobo_property_bag_batch_pop (background);
```

This would allow the listener code to batch all the expensive requests into a single pass.

Configuration handlers Consider the example above, we would like to be able to change properties on the system and have those properties to take effect independently of whether a listener is registered or not.

A property handler might register with the configuration moniker to be launched when a property changes. This could be done in a file installed in a special location.

6.3 The GNOME VFS becomes deprecated

The GNOME VFS provides an asynchronous file-system interface abstraction that can be used to access local files, remote files, files in compressed files and more.

The problem with the GNOME VFS is that it is very limited: it can only expose a file system like

interface to its clients (very much like the Unix interface after which it was modeled).

As covered previously, monikers define an object naming space, and monikers can be defined for any type of resource that the GNOME VFS supports (a transitional path might include a set of monikers implemented on top of the actual GNOME VFS).

A file dialog could request a moniker to be resolved against a "Graphical File Listing" interface, which might result in a miniature Nautilus window to be embedded in the dialog box.

It would be possible to entirely reuse the existing GNOME VFS code by providing monikers for the various access methods that would handle the special cases "Stream", "Storage" and "FileListing". Other interfaces will be plugged into the moniker handler to support the richer content.

For instance, consider the `trashcan:` moniker. The trashcan moniker could be resolved against various interfaces. A file manager would resolve it against a DirectoryListing interface to display the contents of it; It could resolve it against a "Control" interface to get a trahscan custom view (to configure the values in the trashcan); a PropertyBag interface could be used to programmatically configure the various settings in it.

7 Other monikers

There is another family of moniker handlers that are worth stuyding. The filtering moniker handlers and the caching moniker handlers.

7.1 The `streamcache:` moniker

The idea of the `streamcache:` moniker is to be basically a shared library moniker handler that provides a cache for the IDL:Bonobo/Stream:1.0 interface.

This moniker is very simple, during resolution it requests the IDL:Bonobo/Stream:1.0 interface from its parent and it can only expose the IDL:Bonobo/Stream:1.0 interface to clients.

The plus is this: it is a shared library component, which will run in the address space of the application that will use the Stream, and it provides a cache to the parent Stream (so we can use small granular method invocations, and the stream cache can do the traditional buffering).

Think of this difference as the one between an application using write()/read and the application using fwrite/fread/getc/putc: although many applications can implement their own buffering, most of the time just using the libc-provided ones (fwrite/fread/getc/putc) will do it. This is exactly what the `streamcache:` moniker will do: By appending this to a stringified representation of a moniker, you can get a stream cache for free.

7.2 The `#gunzip`, `#utar` filtering monikers

The `#utar` moniker is a moniker that would implement tar file decoding (the same concept can be used for other archive formats). This moniker uses an auxiliary tar component handler. The moniker connects the tar component handler to the parent object's Stream interface and returns the resulting object. The result of the `#utar` moniker can be either a Bonobo/Stream (for a file reference) or Bonobo/Storage (for a directory reference).

For example:

```
file:/home/miguel/mail-backup.tar#utar:2000/may/1001
ftp://ftp.helixcode.com/pub/sources/gnome-libs-1.2.tar.gz#gunzip#utar:/README
```

The beauty of this system is that if two applications use the same moniker, they would be sharing the same data without having to uncompress two times the same tar file.

This is all achieved transparently. This would happen in quite a few instances, for example, if you are exploring a compressed tar file in a file manager and you drag the file to another Moniker-aware application, say Gnumeric, Gnumeric would be using the same file that was openened by the file manager instead of having two uncompressed sets of files in your system.

The above scenario is particularly useful if you have little space, or if the process of untaring a file would take a long time.

7.3 The `propertycache:` moniker

Accessing individual properties over and over might take quite some time due to the CORBA round trips. The `propertycache:` moniker would be also a shared library handler that would basically activate the property moniker, and would set up property listeners (which would be notified of changes in the property data base).

So if your application does a lot of queries to a property, you might just want to append this to improve performance and not care about doing clustered reads, the cache would do this for you.

This is not implemented, as it requires the property moniker to be written.

8 The accidental invention

Monikers were invented originally in OLE2 to implement Object Linking. The OLE2 programmers accidentally invented an object naming system.

This object naming system is not only very powerful, but it is extensible and it helps make the system more consistent.

9 Monikers and the GNOME VFS

Some people ask: monikers look as if they are just re-implementing the GNOME-VFS, why is that?

For a storage backend you can always use something like bonobo_storage_new ("gnome-vfs") and get away with life.

The main difference between the gnome-vfs, and monikers is that monikers are used to implement an object-based name space, while the gnome-vfs is a fine abstraction for naming files and directories. The moniker space goes well beyond this.

When Ettore, Nat, and I designed the GNOME VFS in Paris, Ettore had a grander vision than Nat or I had. Nat and I wanted exactly what the GNOME VFS is: an asyncronous, pluggable virtual file system implementation. Ettore wanted something more general, something that would implement an object name space. And some of the design decisions in the core of the gnome-vfs reflect some of this thinking, but the API and the infrastructure was limited to handling files.

A few months later, we finally understood completely the moniker system, and we realized that monikers were an object naming space, and that if done correctly monikers would be able to implement Ettore's initial vision for having an object-based naming space.

10 Open Issues

We will need to research the implementation requirements for asyncronous parsing and resolution of Monikers.

Currently, both the Object Activation Framework and Bonobo support asyncronous activation. Implementing this for Monikers should not be hard, but might require a few changes in the Moniker interface.

11 Conclusion

Monikers are very powerful mechanisms that can unify the name space of objects in the system and can be used to provide a uniform access method for a wide variety of tasks:

- → Component initialization
- → Addressing objects
- → Addressing sub-objects in a compound document.
- → Implementing Object Linking.
- → Implementing nested objects, and nested handlers for file systems.

12 Acknowledgements

The Bonobo moniker implementation was done by Michael Meeks.

The design for the Bonobo moniker system was done by Ettore Perazzoli, Michael Meeks and myself.