
Repository and Tools

Guide and Tutorial
version 0.2

Jörn W. Janneck[#], Robert Esser^b, Kim Mason^d, Mark Jacobs^b

Draft

Moses Memo
February 2002



University of California at Berkeley[#]
Adelaide University^b
Swiss Federal Institute of Technology^d

Contents

1	Introduction	3
2	Extending a Repository-based System	3
3	The Repository and Name-space	3
3.1	The name space: paths and directories	4
3.2	Retrieving and storing objects	4
3.3	The FileSystemRepository	4
3.4	Listeners	5
4	The Type System	5
4.1	Datatypes	5
4.2	Conversions, relations and the datatype system	6
4.3	Asserted types, implied assertions, reachable types, and conversion paths	6
5	Tools	8
5.1	The ToolFactory type	8
5.2	The ToolFactory interface	8
5.3	Tools and tool factories	9
5.4	Standard Repository User Interface	9
5.4.1	Repository Browser	10
5.4.2	Repository Tree	10
5.4.3	Tool Pane	10
5.4.4	Parameter List	11
5.4.5	Tool Bars	11
6	Tutorial: extending the system	11
6.1	Adding a new data type	11
6.1.1	Adding the <i>ROM</i> , <i>RosettaML</i> and <i>Rosetta</i> types into the standard type system.	11
6.1.2	Creating a Java class to implement the relation between <i>ROM</i> and <i>RosettaML</i>	14
6.2	Integrating a tool	16
6.3	(.	16
6.3.1	Creating an abstract generic tool factory	17
6.3.2	Storing and retrieving repository objects	18
A	Predefined data types	21
B	Predefined tools	21
B.1	Repository Management Tools	21
B.2	MOSES Tools	22
B.3	Other Tools	23
C	Configuring the repository and the tools	24
C.1	Repository configuration variables	24
C.2	Moses tools configuration variables	24

1 Introduction

MOSES is an integrated, extendable tool suite for specifying concurrent systems with a range of modeling formalisms. The tool suite includes a set of tools for visual programming in which visual languages (formalisms) are defined. System components are modeled using (graphical) editing tools and executed either as a stand-alone program or within an animation framework.

A major design objective of the MOSES tool suite is to make it easy for users to extend the system, and to use parts of the system in new and unforeseen ways. A key *application programming interface* (API) of MOSES is the *Repository*. The Repository provides the functionality for persistently storing typed objects in a hierarchical name-space. The type of such an object determines (both directly and indirectly) the set of *tools* which can be applied. Typically, objects could be representations of complete systems or components of a system. Their type could identify the kind of formalism used in the specification (e.g. Petri nets, process networks, etc.). The hierarchical name space could map directly onto a machine's file system, web server, etc.

The tool infrastructure adds to this by providing tools and mechanisms to organize tools, to configure them, and to configure the work space of a user. It also provides all the basic tools for manipulating objects inside a repository.

2 Extending a Repository-based System

Users typically extend a repository-based system in the following ways:

1. Adding new kinds of data structures. Most applications will be manipulating 'structured' data, rather than raw bytes or even raw text strings. They will use graphs, DOM trees, images, lists, abstract syntax trees, graph types, etc. Users will want to add to these data structures, which might raise the issue of making instances of them persistent. The conversion/datatype infrastructure is designed to make this task as easy as possible.
2. Adding new operations on data (tools). When writing new tools that manipulate repository data, typical issues are the whether a tool can operate on a given set of data objects, and how it can store and retrieve objects. Also, tools need to be configured, organized and integrated into some kind of environment. The repository and tool infrastructure provides simple but powerful and extensible mechanisms for doing all this.
3. Extending the repository itself. Advanced users may want to implement repositories that do not store their data in a file system, or at least not in the straightforward fashion the basic `FileSystemRepository` does it. Examples might be a database-based repository, a networked multi-user repository, or even a RAM-based repository. Tools, the ways they interact with the repository, and the kinds of operations that can be executed on repository objects, should be unaffected (as far as possible) by these representational issues.

3 The Repository and Name-space

A repository is defined by the `moses.rep.Repository` interface. The job of the repository is to provide a mechanism for storing and retrieving objects (i.e. Java objects) under a *name*. These names are structured in a *name space*, which is hierarchical, very much like the directory tree of a file system.

Objects are identified to the repository to be of a certain *datatype*, or just *type* for short. The repository uses this information to store the object in some representation that is appropriate, both to the kind of object and the way the repository is organized. Because the repository may not be able to store the object in its original form, it may need to convert between the object and some representation of it that can be made persistent. Likewise, when retrieving objects, it may have to do transformations between the storage representation of an object and the type the application is interested in.

A key functionality of the repository is to maintain datatype information about objects, and to use this information in order to convert between storage representations and application-level datatypes.

3.1 The name space: paths and directories

Like simple file systems, the name space administered by a repository is strictly hierarchical, where a full name, called a *repository path* or just *path*, of an object consists of a (possibly empty) sequence of (non-empty) **Strings**. A path is represented as an instance of the class **RepositoryPath**. As is usual, paths are represented by separating their elements with a dash, i.e. the path consisting of the names "abc", "def", and "ghi" would be written as "/abc/def/ghi", where leading or trailing slashes may be added or omitted. The *length* of a path is the number of **Strings** in it.

In the following we will introduce some terms to describe parts of paths and relations between paths. We say that a path *p* *prefix* of *q*, if *p* is fully contained as the beginning (the left) of *q*. So "/a/b/c/" has prefixes "/" (the *empty* path), "/a", "/a/b", and "/a/b/c", i.e. a path is always a prefix of itself. *p* is a *proper prefix* of *q* if it is a prefix of *q* and shorter than *q*. It is the *direct prefix* of *q* if it is a proper prefix that is exactly one element shorter than *q*—"a/b" is the direct prefix of "/a/b/c". The *last* element of a path is the rightmost **String** in it.

Note that the paths themselves are purely syntactical devices, the **RepositoryPath** class does not give them any meaning. It is therefore also not necessary, to distinguish between things such as relative or absolute paths at this point. These are distinctions that tools may make for the paths they encounter.

3.2 Retrieving and storing objects

Objects are retrieved from the repository using a **getAs** method, the most elaborate form of which allows the specification of the desired datatype, and the path, as well as a **Map** object containing additional information that is passed into the conversions.

The last item may be used by an application to provide additional information that may help the converters to do their job. The repository will compute possible conversion paths between the current storage type of the object and the requested datatype. If that datatype is not reachable for the object (i.e. if there are no conversion paths), an exception will be thrown.

If the current datatype system finds more than one conversion path, it chooses one of them according to criteria specific to the repository implementation. Usually, injective and reversible paths should be preferred.

Storing an object into the repository is done using one of the **putAs** methods, which also requires the specification of the object itself, the path it is to be stored under, and the object's datatype. Like **getAs**, one may specify a **Map** containing data that is passed to the converters. Furthermore, some **putAs** methods accept a **viaTypes** parameter, which is a **List** of datatypes that is used to control the selection of the conversion path. The 'valid' conversion paths must contain all the types in that list, in the order in which they appear in the list. For example, when storing an **Image** object into the repository, there may be more than one conversion path for that image, depending on the format it is stored in (say, **JPEG** and **GIF**). In order to control the format, one may pass a list containing the **JPEG** datatype to the **putAs** method. The **putAs** methods automatically asserts all intermediate types in the conversion path that was chosen.

Another way to store an object into the repository is via an **updateAs** method call. These methods are similar to **putAs**, except that they retain further existing type information if there was already an object present at the path. For example, assume we load an object that is an **XML** document into a text editor. It has types **XML** and **Text** asserted for it. Using **putAs** to store it to its old location after editing would essentially overwrite the object, and because the editor has no knowledge about the object being also an **XML** file, simply stores it as **Text**, thus effectively removing **XML** from the asserted types for that object. By contrast, **updateAs** retains all additional type information.

3.3 The FileSystemRepository

The class `moses.rep.FileSystemRepository` implements a repository on top of a hierarchical file system in a very straightforward manner, i.e. objects are represented as files, directories in the repository correspond to directories in the file system. The type information for all objects in a directory is stored in a special file ("*.typemap*") which is an XML-representation of a **Map**-based data structure, mapping file names to collections of asserted types. Moving directory structures in a repository tree using 'normal' file operations should not cause any problems (except of course in cases where the applications depend on the specific location of a directory, such as directories containing tools, which are registered in an environment path variable).

The storage type of all objects (except directories) is **Bytes**.

The `FileSystemRepository` is built by extending `moses.rep.AbstractRepository`. This abstract class exists to make it easier to implement new repositories by taking care of most administrative issues (type assertions, object type conversions, consistency and parameter checks, listener notification). Concrete subclasses need only implement a small set of methods that essentially handle the physical storage of data.

3.4 Listeners

A repository allows listeners (instances of `moses.rep.RepositoryListener`) to be registered with it, which provide a simple mechanism to monitor repository modifications. Listeners are called for the following events in the repository:

- when objects are copied or inside the repository,
- when objects are deleted from the repository,
- when new objects are added to it or updated,
- when the type assertions for an object change,
- when the repository is closed.

Note that directories are just a kind of object.

4 The Type System

4.1 Datatypes

A *datatype* (or *type*) is a set of (Java) data structures, of networks of objects. Datatypes may be very simple (such as **Bytes**, which is the set of all **byte** arrays, or **Text**, which is the set of all **Strings**) or arbitrarily complex (such as **Collection**, the type of all **Maps**, **Sets**, and **Lists** (with a few constraints on the things they may contain), or **AttributedGraph**, containing all MOSES model graphs).

These datatypes may or may not coincide with Java classes/types. Some do, such as **Bytes** and **Text**, others span instances of several classes (such as **Collection**), yet others may only allow for some instances of a specific class (such as **Java**, which describes all **Strings** that are Java class descriptions). It is best to think of a datatype as a conceptual definition of a kind of data structure. When defining datatypes, no code need be provided that actually determines, for any given object, whether it in fact constitutes a valid instance of that type.

In fact, datatypes themselves are nowhere formally *defined* in the context of the repository. To the repository, they remain fully abstract, they are basically *names* for kinds of objects. The repository associates with each object (a) a *storage type*, i.e. the datastructure in which the object is made persistent (**Bytes** in the case of the `FileSystemRepository`), and (b) a set of *asserted types*, i.e. data structures the object may be converted into, provided that an appropriate conversion exists.

Datatypes are represented by objects whose classes implement `moses.rep.datatypes.Datatype`, the main function they perform is to determine equality. Currently, there are two classes implementing this interface:

- `moses.rep.datatypes.SimpleDatatype`—This class represents the usual object types by a simple name, which is just a **String**. Two instances of this class are equal if their names are.
- `moses.rep.DirectoryDatatype`—Instances of this class represent the type of a repository directory, i.e. instances of classes that implement `moses.rep.RepositoryDirectory`.

Datatypes are required to be able to represent themselves as a **String**. Datatypes can be created from these **Strings** using `moses.rep.datatypes.Util.parseDatatype`.

4.2 Conversions, relations and the datatype system

As mentioned above, the repository maintains for each object the type in which it is physically stored (its *storage type*, as well as a set of *asserted datatypes* that it may be converted into. The pieces of code that perform the actual conversion are implementations of `moses.rep.datatypes.DatatypeConversion`. This interface provides methods that identify the domain and codomain datatype of the conversion, as well as the actual conversion method. It also contains methods that characterize the conversion as being *defined* or *undefined*, *total* or *partial*, and *injective* or *not injective*, *trivial* or *non-trivial*.

If a relation is undefined, its conversion method will always produce an exception. A trivial conversion is one that does not actually change the object to be converted.¹ If it is total, the conversion will be able to convert any object of its domain into one of its codomain, otherwise, only some objects may be validly converted, while others may lead to undefined results. Finally, if a conversion is injective, it produces distinct elements of its codomain for different elements of its domain—an important consequence of this is that the conversion may be reversed without loss of information (so an injective conversion may be considered lossless).

Conversions range from simple do-nothing operations (converting a **Text** object to a **Java** object does not change the object itself) to complex parsing (**XML** to **DOM**), evaluation (**ElanExpression** to **ElanScriptedObject**), or decoding operations (**JPEG** to **Image**).

In the *datatype system* of a repository (an instance of `moses.rep.datatypes.DatatypeSystem`), conversions always occur in pairs within a *datatype relation* – an implementation of `moses.rep.datatypes.DatatypeRelation`. A relation defines the way objects of two datatypes can be converted into each other. Users may add and remove relations to and from the datatype system, but usually an application using a repository should provide some form of controlled configuration facility for this.²

One important kind of relation is the *subtype relation*, (`moses.rep.datatypes.SubtypeRelation`). This defines the kind of relation between two datatypes such as **Text** and **Java**, i.e. where the physical object representation is identical (thus the actual conversion functions are identities), but one datatype consists of only a subset of the objects in the other datatype. The difference between the two conversions is that the one from the supertype to the subtype is total, while the inverse is not.

In any given state, the datatype system can be considered as a graph structure whose nodes are datatypes, and whose edges are (undirected) relations, each of which is made up of two directed conversions even though a conversion may be 'undefined', which basically makes the relation directed. We will also call this the *datatype graph*, or *type graph* (see figure 1).

4.3 Asserted types, implied assertions, reachable types, and conversion paths

In the repository, each object is stored as some datatype, called its *storage type*, which is chosen by the repository and depends on its implementation.³ When retrieving this object, applications are usually interested in some other representation, so the object must be converted from its storage type to these other datatypes. The repository looks for a chain of conversions (there may not be a direct conversion) that starts with the object's storage type, and ends with the type requested by the application.

The datatype graph essentially defines the set of datatypes that an object may be converted into, its *reachable types*. It does this by employing the following recursive definition of the set of reachable types:

- The storage type of an object is reachable.
- If t_1 is reachable, and there exists a total conversion from t_1 to t_2 , then t_2 is also reachable.
- If t_1 is reachable, and there exists a partial conversion from t_1 to t_2 , and t_2 is an *asserted type* for the object, then t_2 is also reachable.

Note that this defines the role of *asserted types*, which is a set of datatypes the repository keeps along with each object. Essentially, when the repository needs to employ a partial conversion to convert to a type, it cannot

¹Conversions between a type and a proper subtype are usually of this kind. For example, the conversions between **Text** and **XML** are both trivial, but one is total (every **XML** object is a valid **Text** object), while the other is not.

²The MOSES repository browser expects a datatype system configuration object at the path `/config/DatatypeSystem` that it then feeds into `moses.rep.datatypes.Util.configureDatatypeSystem`.

³A repository may choose different storage types for different objects, depending on the object's type and/or its location.

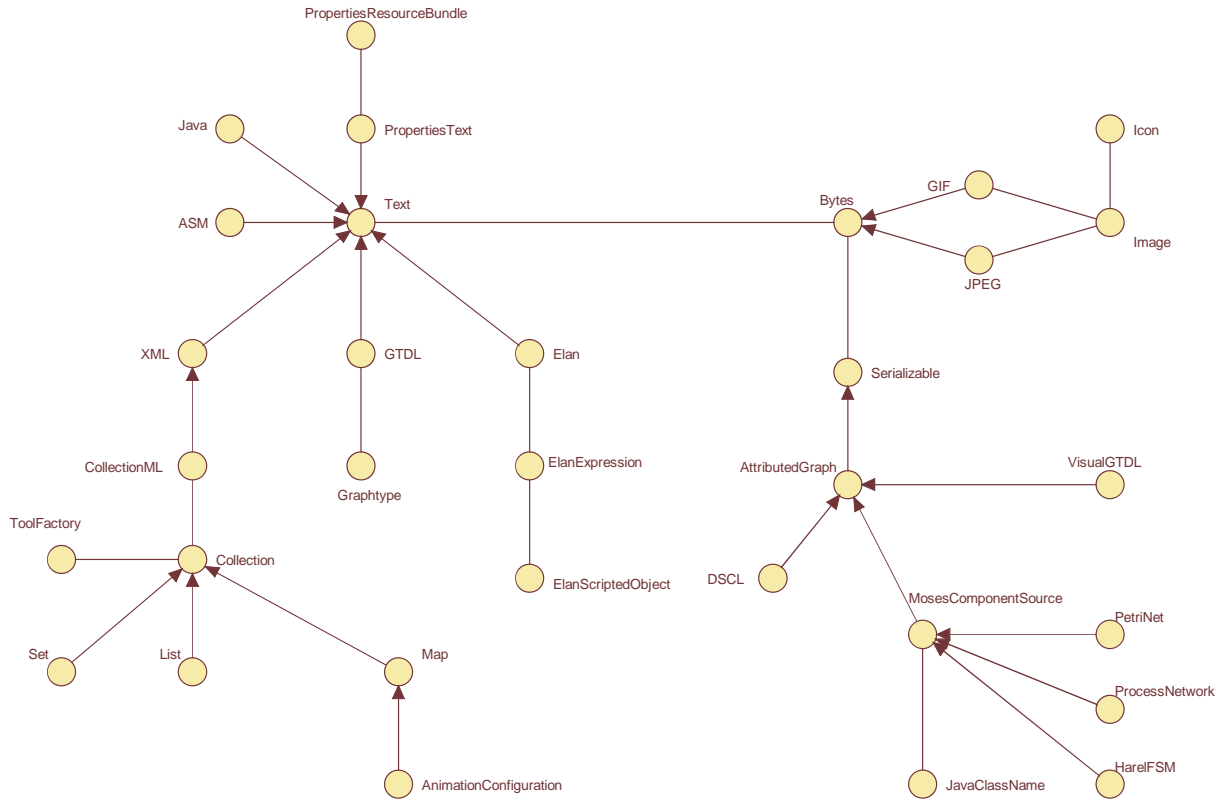


Figure 1: The standard MOSES datatype system.

be sure that this conversion will work, since it may not be defined for the object in question (which is what partial means). In this case, an asserted type serves as a guarantee that the object may in fact be legally converted to that type.

Example. Consider an object stored as **Bytes**, which we want to retrieve as an **Image**. Assume there are two conversion pertinent to this, viz. **Bytes** \rightarrow **JPEG**, and **JPEG** \rightarrow **Image**. Here **JPEG** is a subtype of **Bytes**, since clearly not every collection of bytes represents a valid image in JPEG format. Thus, the first conversion is partial. However, every valid JPEG description may be transformed into an image, thus the second conversion is total.

In order to reduce the number of explicit assertions required to perform a conversion, we define the notion of *implied assertion* as follows: If T_1 is asserted (explicitly or implicitly), then this implies the assertion of T_2 iff (a) the conversion from T_2 to T_1 is defined and trivial, and (b) the reverse conversion from T_1 to T_2 is defined, trivial, and total.

Example. Consider an XML document, which is stored as **Bytes**. The conversion path from **Bytes** to **XML** is normally **Bytes** \rightarrow **Text** \rightarrow **XML**. Since both conversions are partial, we require both **Text** and **XML** to be asserted for that object. However, since the assertion of **XML** implies the assertion of **Text**, the user need only provide one type assertions, as opposed to two, one of which seems somewhat redundant.

Following the above algorithm without any asserted types, the only reachable type would be **Bytes**. Asserting the type **JPEG** for this object, however, makes both **JPEG** and **Image** reachable. On the other hand, Asserting **Image** rather than **JPEG** does not help us, because the conversion from **Bytes** to **JPEG** cannot be performed since it is partial.

The paths from the storage type to each reachable type through the type graph, via defined conversions, are the possible *conversion paths* to that type.

5 Tools

The repository, as described in the previous sections, is an infrastructure for managing typed information. This functionality may be embedded within an application as is, but many applications will use the repository for storing artifacts generated as a result of user actions. Here the repository will be viewed via a user interface allowing users to visualize and manipulate repository objects via a set of tools. Tools themselves typically only operate on a subset of all possible object types and may also have a set of options that are used to configure it upon instantiation.

Given the above outline of a tool it can be seen that it makes sense to also have tools represented by repository objects. The following sections describe how tools are represented in the default repository user interface and how they are created, manipulated and applied to repository objects.

5.1 The ToolFactory type

As can be seen in figure 1 the *ToolFactory* type is related to the *Collection* type. A conversion is provided between objects that implement the Java interface `moses.tools.ToolFactory` and *Collection*. A *ToolFactory* is a simple specific map of key-value tuples. The only mandatory key is the *ToolClassName*. The value is a string representing a fully qualified *Java* class name that implements the *ToolFactory* interface.

Other keys include *TOOLCOMMENT*, *TOOLIMAGE*, *TOOLOPTIONS*. The associated values are used to either enhance the visualization of the tool object in the repository or to store tool specific options. The conversion associated with the *ToolFactory-Collection* will produce an instance of the *Java* class specified by the *ToolClassName* key.

5.2 The ToolFactory interface

The *ToolFactory* interface is used by the repository user interface to extract information about repository tools. The following code describes the methods in this interface and their use.

- `public void initialize(Map options);`
This method is called once before *canCreate* or *create*. It configures the *ToolFactory* using the parameter *options* – a map from strings to collection-compliant data objects.
- `public boolean canCreate(Map params);`
This method is called to ascertain whether the tool is applicable to the map of parameters. By default, *params* contains a *List*, key 'arguments'. This list of arguments is usually identified by the currently selected repository objects. A return value of *true* indicates that the tool can be applied to these parameters.
- `public boolean create(Map params) throws RepositoryAccessException;`
This method should be called to create the tool. If the parameters are un-acceptable, *false* is returned, and the tool is not created.
- `public String getInfo();`
A simple method that returns some information about this tool that can be displayed in a user interface
- `public Icon getIcon();`
This method returns an *Icon* that represents the tool. *null* indicates that no special icon is defined.
- `public Map getOptionDescriptions();`
Returns a map of option names and types or *null* if none.
- `public Map getOptionValues();`
Returns a map of option names and values or *null* if none.

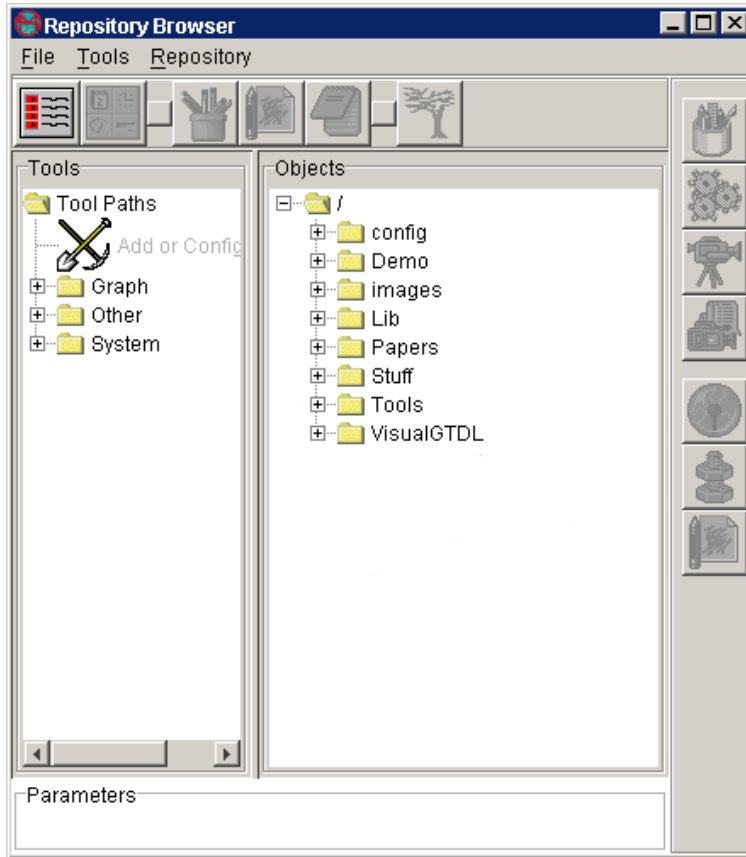


Figure 2: The standard MOSES Repository User Interface.

5.3 Tools and tool factories

A class implementing the *ToolFactory* interface represents not a specific tool but a set of specific tools. In other words an object in the repository of type *ToolFactory* represents a specific instance of a class implementing the *ToolFactory* interface. This distinction is useful where a *ToolFactory* also defines a set of options that may be different between tool instances.

An example of this is the *TextEditorToolFactory*. This factory defines an option *Elan Checking Expression* into which an *Elan* expression can be defined that is used to check the validity of the contents of the edit buffer. Hence it is possible for the editor to give the user a visual feedback regarding the validity of any file being edited. As a result it is simple to define a number of text editing tools for different languages each defining a different checking expression.

5.4 Standard Repository User Interface

The MOSES tool-suite is a modeling and simulation environment in which artifacts are stored in a repository. This repository also contains a user interface which is independent of the MOSES tool-suite. As a result it can be used in many other contexts none of which depend on MOSES. To extract only the repository and its user interface only the `moses.rep` and `moses.tools` packages and sub packages are required.

In this section a brief overview of the user interface is given. Figure 2 shows the *Repository Browser* with all panes open. Each pane captures a particular view on the repository.

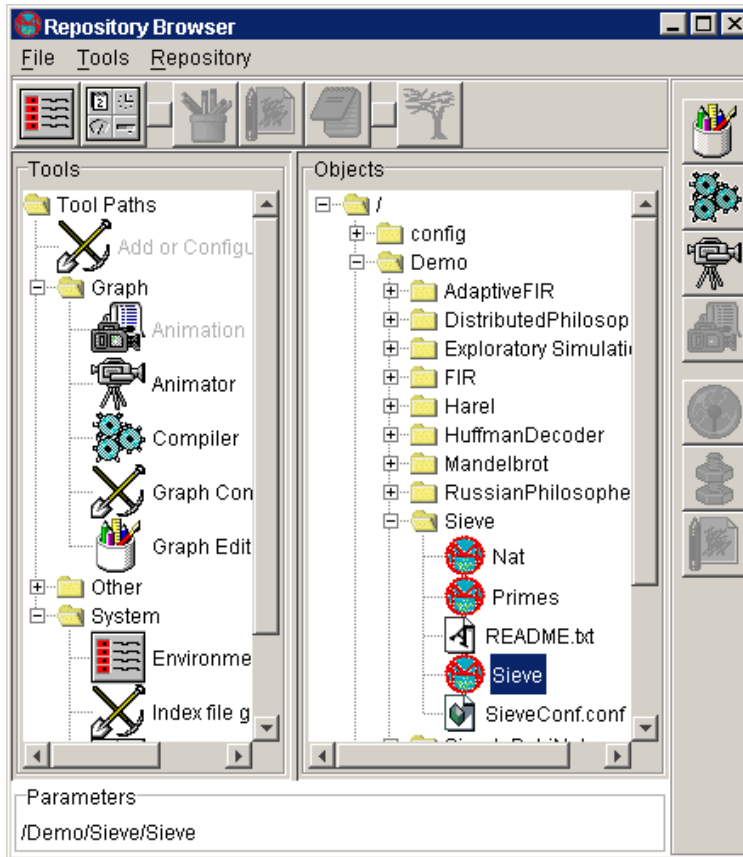


Figure 3: The MOSES Repository User Interface showing the effects of selecting an object.

5.4.1 Repository Browser

The Repository browser is the container for a number of graphical objects. It extends `javax.swing.JInternalFrame` and as such can be inserted either into a Window or into a desktop⁴. The browser has a number of menu items that allow repositories to be opened/closed and the various panes to be configured.

5.4.2 Repository Tree

The Repository Tree is the *Objects* pane shown in figure 2. It gives a hierarchical view of the currently open repository. A folder icon with a + sign next to it indicates that there are objects contained within it. By selecting a folder or an object will cause the parameter repository path to be shown in the *Parameter List* and all applicable tools as enabled in the *Tool Pane* and *Tool-bars*.

5.4.3 Tool Pane

The tool pane shows all repository tool objects in the set of current tool paths. Tool paths may be edited by selecting the *Tools-Path* menu entry. In this way a subset of tools may be made visible in the tool pane. As noted above the individual tools will be enabled or disabled depending on whether they are applicable to the current selection in the repository tree. The tool pane may be hidden by selecting the *Tools-Hide/Show Tool Pane* menu entry.

⁴This description does not cover the issues of programming user interfaces in Java.

5.4.4 Parameter List

The parameter list is used to list the repository paths of current repository tree selection. Of occasional use is the ability to copy the parameter text by selecting the required text and using the standard platform *copy* keyboard short-cut. This text may be used to enter a path into a configuration dialog.

5.4.5 Tool Bars

Upon loading a repository as many tool-bar slots are created in the repository browser as there are tool-bar entries in the `/config/RepositoryBrowser/ToolBars` repository folder. Tool bars may be shown/hidden via the *Tools/Tool Bars* menu entry. They may also be dragged to one of 5 positions: North, South, East and West of the main panes and also into a separate window. The positioning and whether a tool bar is visible is stored in the specific tool bar repository object in the `/config/RepositoryBrowser/ToolBars` repository folder.

Individual tools may be deleted from the tool bar by dragging the tool bar button onto itself, repositioned in the tool bar by dragging a tool bar button onto another position in the tool bar and added to the tool bar by dragging a tool from the tool pane into the tool bar.

6 Tutorial: extending the system

In this mini tutorial it will be shown how the repository can be configured to be an integral part of a new tool-suite. As an example, a tool environment will be provided to support Rosetta, a Systems Level Design language. Specifically the type system of the repository will be extended to support types relevant for a Rosetta environment.

6.1 Adding a new data type

Figure 4 shows part of the standard type system with provision made for three new types: *Rosetta*, *ROM* and *RosettaML*. Each type represents a class of artifacts that are required to be stored and retrieved by *Rosetta* tools. The type *ROM* represents a Rosetta object, called the Rosetta Object Model. This type is used by Rosetta tools, but can only be used for temporary storage. The type *RosettaML* represents a format for the ROM that is permanent and interchangeable, as it is a subtype of *XML*. The type *RosettaSource* represents the textual representation of the Rosetta class. Objects of this type are converted using a Rosetta compiler into object code for simulation. Also it can be seen in figure 4 that *Rosetta* is a subtype of *Text* whereas a relation exists between *ROM* and *RosettaML*. In the figure it can be seen that this relation is implemented by the Java class `moses.rep.datatypes.lib.ROMRosettaMLrelation`.

Integrating the three new types into the repository consists of the following 2 tasks:

- Adding the *Rosetta*, *RosettaML* and *ROM* types into the standard type system.
- Creating a Java class to implement the relation between *ROM* and *RosettaML*.

6.1.1 Adding the *ROM*, *RosettaML* and *Rosetta* types into the standard type system.

Upon a repository being opened, the type system is loaded from a specific repository object `/config/DatatypeSystem`. This object is an instance of *Collection* and contains a list of relations and subtype relations. There are two ways in which this can be modified:

- **Direct modification of `/config/DatatypeSystem`**

The `/config/DatatypeSystem` repository object is a collection of type relations and subtype relations represented as a list. As this object is represented in *CollectionML*, an XML language, which in turn is a subtype of *Text* it can be directly edited with a text editor tool. To achieve the same result as that shown in figure 4 4 entries need to be made into the list – one for the subtype of Text (*Rosetta*), one for the subtype of Serialization (*ROM*), one for the subtype of XML (*RosettaML*) and one for the relation between *ROM* and *RosettaML*.

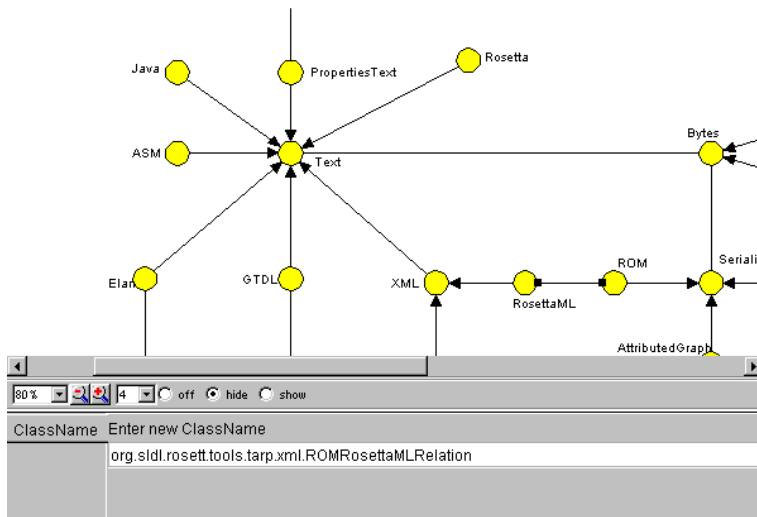


Figure 4: The standard type system graph extended with *ROM*, *RosettaML* and *Rosetta* types.

The following fragment of XML shows the top-most list of the Collection. Inserting a relation or subtype relation is simply a matter of inserting an entry references in the form of `<ref idref="G100"/>` where the *idref* has not been previously used. Insert four additional entries into the top-most list of the Collection.

```
<?xml version='1.0' standalone='yes'?>
<!-- a collection data structure -->

<collection version="0.1" root="G0">
  <list id="G0">
    <ref idref="G1"/>
    <ref idref="G2"/>
    <ref idref="G3"/>
    <ref idref="G4"/>
    <ref idref="G5"/>
    <ref idref="G6"/>
    ...
  
```

Now that an additional four entries have been inserted into the Collection they must now be defined. To define a relation the entry is just a reference to a string representing the fully qualified Java class name that implements the relation. e.g.

```
<text id="G100">moses.rep.datatypes.lib.ROMRosettaMLRelation</text>
```

Slightly more complex is the definition of a subtype. Rather than it being a reference to a string, it is a reference to a list that in turn contains references to 3 elements: the label "*SubtypeRelation*", the *subtype* name and the *parent type* name. For example consider the XML representing a subtype relation:

```
<list id="G101">
  <ref idref="G35"/>
  <ref idref="G200"/>
  <ref idref="G43"/>
</list>
<text id="G200">Rosetta</text>
```

In this case "G35" is a reference to the string *SubtypeRelation* i.e. `<text id="G35">SubtypeRelation</text>`. The second entry "G200" is a reference to the string containing the name of the subtype – *Rosetta*. The last entry is a reference to the string representing the parent type – in this case *Text*.

Three subtype relations will need to be entered to complete the modification. Enter the following XML into the `/config/DatatypeSystem` file. In this case "G53" is a reference to the string *ROM*, "G55" is a reference to the string *XML*.

```
<list id="G101">
  <ref idref="G35"/>
  <ref idref="G200"/>
  <ref idref="G43"/>
</list>
<text id="G200">Rosetta</text>
<list id="G102">
  <ref idref="G35"/>
  <ref idref="G201"/>
  <ref idref="G53"/>
</list>
<text id="G201">ROM</text>
<list id="G103">
  <ref idref="G35"/>
  <ref idref="G202"/>
  <ref idref="G55"/>
</list>
<text id="G202">RosettaML</text>
```

• Using the MOSES graph editor

A repository's type system can be represented by a graph. This graph is another example of what in MOSES is referred to as a formalism. In this case the formalism, called *Datatype System Configuration Language* (DSCL), defines vertices to be types and arcs to be either representing a subtype relation or a general relation where an arc attribute is used to define the implementing *Java* class. The MOSES graph editor is then used to edit the graph representing the type system. Once the type system has been edited and saved the DSCL Compiler tool is used to translate the graph into the *CollectionML* object `/config/DatatypeSystem`. The following step by step guide will add support for the three Rosetta types *Rosetta*, *ROM* and *RosettaML*:

- In the standard MOSES release the data type system graph is represented by the `/config/Datatype System Configuration`. Select this object and the MOSES graph editor tool in either a tool-bar or the *Tools* pane. You should now have a pane visible similar to that in figure 1.
- In the editor select the *data type* node (yellow circular icon) in the tool-bar. Insert a data type node in the vicinity of the *Text* node.
- Now insert another node in the vicinity of the *Serialization* node.
- Insert another node in the vicinity of the *XML* node.
- Select the pointer icon in the tool-bar and then select one of the nodes you just inserted in the editor canvas. Now select the *Open/Close attribute editor* icon from the tool-bar. An attribute editor pane for the selected node should now be visible.
- Enter the name of the node, e.g. *Rosetta*. Press the *Commit* button in the attribute editor.
- Name the second node you inserted *ROM*.
- Name the final node *RosettaML*.
- In the tool-bar select the *subtype relation* button and click on the *Rosetta* node followed by clicking on the *Text* node. A subtype relation has now been inserted.

- Draw another subtype relation between the *ROM* and *Serialization* nodes.
- Draw a final subtype relation between the *RosettaML* node and the *XML* node.
- In the tool-bar select the *relation* button and click on the *ROM* node followed by clicking on the *RosettaML* node. An unnamed relation has now been inserted.
- Select the pointer icon in the tool-bar and then select the unnamed relation arc. As the attribute editor is still open the attributes for this arc are shown. Enter *moses.rep.datatypes.lib. ROMRosettaMLRelation* as the fully qualified Java class name of this relation.
- In the *Graph* menu select the *Save and Exit* entry.
- Make sure that the repository object */config/Datatype System Configuration* is selected. Select the *DSCL Compiler* tool.
- In the resulting dialog box select the */config/DatatypeSystem* object and press the *Accept* button.

The data type system has now been updated. However it still needs to be loaded into the repository. To do this in the repository browser select *Refresh* from the *Repository* menu.

Naturally it is far preferable to edit the type system as a graph as it is faster and far less error prone. However it does assume the presence of a large portion of the MOSES environment. Often a type system will not change and it may be possible to do without all of the MOSES tool suite except for the repository specific components. Here any change to the type system could still be made using a text editor as outlined above.

6.1.2 Creating a Java class to implement the relation between *ROM* and *RosettaML*.

This section describes the structure of a datatype relation. In the previous section the type system was extended with a type that had a subtype relationship with the type *Text* and another type that had a more general relation.

Both methods for extending the type system differentiated between a subtype relation and a general relation. It is important to understand that the subtype relation is a specific kind of general relation. In both cases the Java class implementing the relation is derived from *moses.rep.datatypes.GenericDatatypeRelation*. In the case of the subtype relation the class *SubtypeRelation* simply defines a relation with identity conversions. The following code is the actual definition of the *SubtypeRelation* class.

```
package moses.rep.datatypes;

public class SubtypeRelation extends GenericDatatypeRelation {

    // Ctor
    public SubtypeRelation(Datatype subtype, Datatype supertype) {
        super(new IdentityConversion(supertype, subtype, false),
              new IdentityConversion(subtype, supertype, true));
    }
}
```

Here it can be seen how the *SubtypeRelation* class supports all subtype relationships. In general it is necessary to provide more information about the conversion. In the following such a specific relation is described. As before it is taken from the Rosetta example, describing the implementation of the *moses.rep.datatypes.lib.ROMRosettaMLRelation* class. The following code actually defines 3 classes. The public class is a subclass of *GenericDatatypeRelation* while the 2 private classes extend *AbstractDatatypeConversion* each define a unidirectional conversion. Therefore the first fragment defines the *ROMRosettaMLRelation* class and a constructor that refers to the 2 conversion classes: *Rosetta2ROMConversion* and *ROM2RosettaConversion*.

```
package moses.rep.datatypes.lib;

import moses.rep.datatypes.*;
import moses.rep.datatypes.lib.*;
```

```

public class ROMRosettaMLRelation extends GenericDatatypeRelation {

    public ROMRosettaMLRelation() {
        super(new ROM2RosettaML(), new RosettaML2ROM());
    }
}

```

The following code defines a conversion from the objects of type *ROM* to the type *RosettaML*. Not only is a conversion routine defined but also information about the conversion itself e.g. if it is *defined*, *total* and *injective*. The constructor defines the *domain* and *codomain* type respectively of the conversion. Also it is important to note that the conversion is total – all objects of type *RosettaML* are valid *ROM* objects and vice versa.

```

class ROM2RosettaML extends AbstractDatatypeConversion {

    public Object convert(Object a, Map env)
        throws DatatypeConversionException {

        try {
            StringReader stream = new StringReader((String) a);
            StorableInput input = new StorableInput(stream);
            return (Drawing)input.readStorable();
        }
        catch (IOException e) {
            throw new DatatypeConversionException(this, a, "General I/O exception caught: " +
                e.getMessage());
        }
    }

    public boolean isTotal() {
        return true;
    }

    public boolean isInjective() {
        return true;
    }

    public boolean isDefined() {
        return true;
    }

    ROM2RosettaML() {
        super(new SimpleDatatype("ROM"), new SimpleDatatype("RosettaML"));
    }
}

```

Finally the following code defines a conversion from the objects of type *RosettaML* to the type *ROM*. Also it is important to note that the conversion is total – all objects of type *RosettaML* are valid *ROM* objects.

```

class RosettaML2ROM extends AbstractDatatypeConversion {

    public Object convert(Object a, Map env)
        throws DatatypeConversionException {

```

```

try {
    StringWriter stream = new StringWriter();
    StorableOutput output = new StorableOutput(stream);
    output.writeStorable((Drawing) a);
    output.close();
    return stream.toString();
}
catch (Exception e) {
    throw new DatatypeConversionException(this, a, "General I/O exception caught: " +
        e.getMessage());
}
}

public boolean isTotal() {
    return true;
}

public boolean isInjective() {
    return true;
}

public boolean isDefined() {
    return true;
}

RosettaML2ROMConversion() {
    super(new SimpleDatatype("RosettaML"), new SimpleDatatype("ROM"));
}
}

```

6.2 Integrating a tool

Integrating a tool into the repository browser is typically a 2 stage process: the first stage is to create a *Java* class that implements the *ToolFactory* interface; the second requires that the open/save mechanism the tool supports be modified so that objects are retrieved from and stored into the repository.

In the following code we will look at integrating an existing tool to work in the Moses environment. The tool we have chosen is included in the demo directory of the Java SDK. We will be looking at the small Notepad example that consists of two files `ElementTreePanel.java` and `Notepad.java`.

In this example the java files will be edited and placed in the *moses.tools.lib* package. This tutorial only shows parts of the fully integrated tool. The full source code is available from the Moses website.

6.3 (

Adding files to repository) The Notepad tool requires several icons for the toolbars and a resource bundle. These will be added to the repository.

Add the six GIF files from the Notepad source directory to a new directory to directory `/config/Icons/Notepad` in the repository. Now change each repository object so that they are all of type `/emphGIF`.

Edit the `Notepad.properties` file in the resources directory of the Notepad demo. Remove all six occurrences of `resources/`. Now add the file to the repository at the location `/Lib/Properties/Notepad.properties` and change its type to include *PropertiesResourceBundle* and *PropertiesText*.

6.3.1 Creating an abstract generic tool factory

In section 5.1 the concept of the *ToolFactory* is described. All tools that rely on the MOSES repository browser are required to implement this interface. This interface is used to extract information from the tool and to instantiate it. Naturally any correct implementation of this class will define a tool. However it is often easier to extend the abstract class *AbstractGenericToolFactory* that implements much of the common house keeping required by most tools. This is what we will do here.

```
package moses.tools.lib;
/*
 * @(#)Notepad.java 1.19 01/12/03
 *
 * Copyright 2002 Sun Microsystems, Inc. All rights reserved.
 * SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.
 */

import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.io.*;
import java.net.URL;
import java.util.*;

import javax.swing.text.*;
import javax.swing.undo.*;
import javax.swing.event.*;
import javax.swing.*;

//moses imports
import moses.tools.*;
import moses.tools.lib.*;
import moses.tools.gui.*;
import moses.rep.datatypes.*;
import moses.rep.*;
import moses.util.*;

public class NotepadTool extends AbstractGenericToolFactory {

    public NotepadToolFactory() {}

    private final static Datatype collectionType = new SimpleDatatype("Collection");
    private final static Datatype acceptableType = ToolConstants.PLAINTEXTTYPE ;
    private static ResourceBundle resources;
```

The class *NotepadTool* implements a number of methods that are called from methods in *AbstractGenericToolFactory* that directly implements the *ToolFactory* interface. The method *_isValidArgument* returns true if it is happy with the array of arguments passed to the tool. In this particular implementation of the method there must only be one argument and the type of that argument must be *Text*. The commented out code shows how to allow the tool to execute on a directory if the tool has the option to start with a blank document.

```
protected boolean _isValidArgument(RepositoryObject[] args) {
    if (args.length != 1)
```

```

    return false;
try {
    RepositoryObject object = args[0];
    Set types = object.getTypes();
    //// is it a directory
    //if (types.contains(object.getRepository().directoryType())) {
    // return true;
    //}
    return types.contains(acceptableType);//we must be able to get a text representation
} catch (Exception e) {
    return false;
}
}
}

```

The methods *_setupCreate* and *_execute* are called during the instantiation process. The *_execute* method is used to instantiate the Notepad tool. The method also passes it the text object retrieved from the repository, . The comments document the functionality of the method.

```

protected boolean _setupCreate(Map parameters) {
    java.util.List args = (java.util.List)parameters.get(ToolConstants.TOOLARG);
    return args.size() == 1;
}

protected boolean _execute(RepositoryObject[] args, Map parameters) throws Exception {
    try {
        RepositoryObject ro =args[0];
        JInternalFrame tool = new Notepad(ro);
        //as the tool extends JInternalFrame we insert it to the Moses desktop
        launcher.addTool(tool);
    } catch (Exception e) {
        e.printStackTrace() ;
    }
    return true;
}

```

The above *_execute* method instantiates a tool with a user interface. In a similar fashion tools that require no user interaction may simply be instantiated and run. The only point to note is that tools with user interfaces are implicitly executed in their own thread. Tools that require no user interaction will be executed in the repository browser thread and may prevent the browser from responding to user interaction. In this case either ensure that the tool will always execute quickly so that the user is not aware that the UI is blocked or create a new thread in which it is run.

For more examples of implementations of these methods look at the tools in the package *moses.tools.lib*.

6.3.2 Storing and retrieving repository objects

One assumption valid for most repository tools is that objects are retrieved from and stored into the repository. The Notepad tool normally loads and stores a properties resource bundle, icons and the text files from the local file system. All of the access to these objects must be done via the repository and the icons and resource bundle must be placed in the repository.

The code to load the resource bundle is not covered in this tutorial, however it can be viewed in the accompanying source code.

Only a few lines need to be modified to load the icons from the repository, rather than the file system. In the *createToolBarButton* method the following changes should be made:

```

protected JButton createToolbarButton(String key) {
    //URL url = getResource(key + imageSuffix);
    //JButton b = new JButton(new ImageIcon(url)) {
    Icon i=null;
    try {
        RepositoryPath p=new RepositoryPath(/config/Icons/Notepad+
            getResourceString(key+imageSuffix));
        i=(Icon)rep.getAs(p,ToolConstants.ICONTYPE);
    } catch (Exception e) {
        // cant load icon, bad luck
    }
    JButton b = new JButton(i) {
        public float getAlignmentY() {return 0.5f;}
    };
};

```

This uses the repositories *getAs()* method, with the datatype of the object and the objects path as the methods arguments. The *getAs()* method returns a Java object as specified by the object parameter in the method call. In this case an icon is returned.

This handles the retrieving of the icons and the resource bundle. Now lets move on to loading text files from the repository. The code for this was in the previous section, but other parts of the Notepad tool will be shown here.

An option exists in the Notepad tool for the user to choose a file to open. The standard Java methods for prompting the user to open a file do not work in the repository. However the repository does come with library methods that enable the user to be prompted for a repository object to be opened. Filters can be added to only display the necessary datatypes, in this case */emphtext*. The following code is from the internal */emphOpenAction* class in the notepad tool.

```

PathChooser pc = new PathChooser(frame, "Open", rep);
pc.addFilterType(acceptableType );
String text=null;
String title=null;
try {
    if (pc.showOpenDialog()) {
        RepositoryPath newPath = pc.getRepositoryPath();
        ro = new SimpleRepositoryObject(rep,newPath);
        updateRepositoryObject(ro) ;
        text = (String) rep.getAs(newPath,acceptableType);
        title = newPath.toString();
    }
} catch (RepositoryException ex) {
    JOptionPane.showMessageDialog(frame,
        ex.getMessage(), "Unable to Open object",
        JOptionPane.ERROR_MESSAGE );
    return;
}

```

The *PathChooser* class is used to prompt the user to select a repository object. The */emphaddFilterType* method is called to filter out all repository objects that do not have a datatype of text. A repository path is retrieved from the pathchooser and this can then be opened via the */emphgetAs()* method.

When saving objects to the repository it is important to make sure that the associated datatypes of the object are not changed. For example if you edit a repository object of type */emphXML* in a text editor you still want the object to be an XML document. The following code fragment shows how to save an object to the repository and not overwrite its datatypes.

```

class SaveAction extends NewAction {

    SaveAction() {
        super(saveAction);
    }
    public void actionPerformed(ActionEvent e) {
        try {
            Document d = getEditor().getDocument();
            String text=d.getText(d.getStartPosition().getOffset()
                                ,d.getEndPosition().getOffset() );
            rep.updateAs(ro.getPath(),text,ToolConstants.PLAINTEXTTYPE );
        } catch (Exception ex) {
            System.err.println("Exception raised while saving to repository: " +ex);
            ex.printStackTrace() ;
        }
    }
}

```

The `updateAs()` method preserves the repository objects existing datatypes.

Finally when integrating stand-alone tools into the repository we must make sure that closing the tool does not close down the complete environment. In the following method the `System.exit(0);` has been commented out.

```

/**
 * Exits the application.
 */
class ExitAction extends AbstractAction {

    ExitAction() {
        super(exitAction);
    }

    public void actionPerformed(ActionEvent e) {
        //System.exit(0); //don't exit Moses
        dispose();
    }
}

```

A Predefined data types

The following data types are to be found in the standard MOSES version of the repository.

- **Bytes** The base (storage) type of all repositories based on a host file system.
- **Text** A string encoded in the platform's default character encoding.
- **XML** This type represents all instances of an *Extendible Markup Language* (XML) description.
- **CollectionML** This type represents collection objects stored in *XML* format.
- **Collection** A *Java* collection.
- **ToolFactory** A description of an instance of a repository tool.
- **Serializable** A serialized representation of a *Java* object.
- **AttributedGraph** An object that can be edited by the MOSES graph editor.
- **GTDL** A *Graph Type Description Language* object representing a MOSES modelling formalism.
- **GraphType** A compiled *Graph Type Description Language* object.
- **Elan** A type that represents an instance of any *Elan* object.
- **ElanExpression** A type representing an expression written in *Elan*.
- **ElanScriptedObject** A type representing the result of an evaluation of an *ElanExpression*.
- **Image** A type representing a *Java* image.
- **JPEG** An image encoded using the JPEG compression algorithm.
- **GIF** An image encoded using the GIF compression algorithm.
- **Java** A fully qualified *Java* class name.

B Predefined tools

This section aims to give a brief overview of the standard set of tools. These tools are classified into tools that are used to manage objects in the repository and tools that are used in the MOSES context – creating and managing heterogeneous models. The following descriptions actually describe *Tool Factories*. These factories are used to create instances of tools. Instances of a particular factory are distinguished by their parameters. A number of tools have no functional parameters and hence typically only one instance is present in the system. Other tool factories may have a number of instances each specialised for a particular task.

B.1 Repository Management Tools

- **Environment Editor** (`moses.tools.lib.EnvironmentEditorToolFactory`)
The Environment editor is a tool that is used to view and/or modify a set of environment variables. These environment variables are typically used by other tools to store tool-specific information. Also the repository itself maintains the *ToolPoolPath* environment variable that it uses to store the paths on which repository tools are made available in the tool pane.

- **Tool Creation Editor** (`moses.tools.lib.ToolCreatorToolFactory`)
This is the only tool that the standard User interface always inserts into the tool pane. By using this tool other tools can be defined and inserted as repository objects into the current repository. Naturally if a tool is inserted into a current tool path then it will appear as a tool in the tool pane from which it can be dragged into a tool-bar.
This tool requires that a fully qualified *Java* class path to be defined. In addition this class must implement the `moses.tools.ToolFactory` interface. Also an image may be defined that represents the tool in the tool pane and tool bars. Tools external to the Java environment can also be defined. Here it is assumed that parameters objects are extracted from the repository and stored on the file system. Any changes to a file will cause it to be read from the file system and inserted back into the repository. A typical example of such an external tool would be an icon editor. Here the executable is defined and the types of repository objects that the editor operates on – in this case **JPEG** and **GIF** format images.
- **Text Editor** (`moses.tools.lib.TextEditorToolFactory`)
Although a text editor is not strictly necessary to manage a repository, as most repository objects are stored in *XML* format, a text editor is useful to manually view and modify these objects. The text editor also allows an *Elan* expression to be defined that is used to check the validity of the contents of the edit buffer. Hence it is possible for the editor to give the user a visual feedback regarding the validity of any file he/she is editing. This checking function runs asynchronously to the editing of the file and is used in the standard repository to define a special text editor that understands the *Graph Type Definition Language*.
- **Object Type Editor** (`moses.tools.lib.EditObjectTypesToolFactory`)
Every object stored in the repository is associated with a set of asserted types. These types and the types that are reachable from them define the set of types that an object can be read as and indirectly which tools can be applied to the objects. The Object Type Editor is used to create and/or modify type associated name and asserted types of an object. The type field is color coded indicating whether a type is asserted (blue), reachable (green), unreachable (red) or not relevant (grey).

B.2 MOSES Tools

- **Graph Editor Tool** (`moses.tools.lib.GraphEditorToolFactory`)
The Graph editor will open for view/edit any object of type *AttributedGraph*. It is the main tool within the MOSES tool suite and has a large number of features for editing graphs of all types.
- **Graph Configuration Tool** (`moses.tools.lib.GraphConfigurationToolFactory`)
A sub-type of *AttributedGraph* is *MosesComponentSource*. This type is used to identify those graphs that represent MOSES components. These components are typically executable and need to be compiled into an executable form. The graph configuration tool is used to set a number of parameters that affect the generation of code. These include the *class* and the *package* name for which code will be generated. Also the formalism in which the component was described can be modified. Naturally care must be taken when interpreting the graph with another formalism.
- **Model Compiler Tool** (`moses.tools.lib.ModelCompilerToolFactory`)
The model compiler tool also only operates on objects of type *MosesComponentSource*. This tool causes *Java* source code to be generated and compiled into a *Java* class. This process makes use of *class* and the *package* name settings manipulated by the graph configuration tool as well as the following global environment variables: *CLASSDIR*, *DefaultDestinationPath* and *TMP* (see section C.2).
- **Graph Animator Tool** (`moses.tools.lib.GraphAnimatorToolFactory`)
Once a set of components is compiled they can either be executed outside of the MOSES environment or within it. The graph animator tool will attempt to simulate and animate the designated MOSES component. Upon activation a MOSES Animation frame will appear into which a number of views on the instantiated component will be displayed. The compiled representation for the component will be sought in the *ClassPath* environment variable. If the component has parameters then a parameter dialogue will appear in which parameter expressions can be entered. Once the component has been successfully instantiated the user has

control over how the component is to be executed: single step, continuous with animation and continuous simulation without animation.

- **Animator Configuration Tool** (`moses.tools.lib.AnimatorConfigurationToolFactory`)
For components with parameters it is often very useful to define a repository object of type *AnimationConfiguration*. Objects of this type are used to store a components parameters for a particular simulation run. The graph animator tool operates either on MOSES components or animation configuration objects.
- **Datatype System Compiler Tool** (`moses.tools.lib.DatatypeSystemConfigurationCompilerToolFactory`)
The Repository data-type system can be derived from a graph representation. These graphs have the *DSCL* type which the data-type system compiler tool recognizes as a description for a type system. The compiler will prompt the user for an output repository object name. Repositories read their data-type System from a Repository object */config/DatatypeSystem*. Hence overwriting this file will change the type system of the repository (after a Repository-Refresh).

B.3 Other Tools

- **Java Compiler Tool** (`moses.tools.lib.JavaCompilerToolFactory`)
This tool operates on objects of type Java. An instance of this tool has 3 parameters defined: *Source Base*, The base directory where the Java object is written onto the file system as a *.java* file; *Code Base*, the base directory where the resulting class file will be written; *Compile Options*, additional compiler options. Upon execution a status pane will appear in which any information from the Java compiler will be written.
- **Repository Importer Tool** (`moses.tools.lib.RepositoryImporterToolFactory`)
Instances of this tool will allow old style MOSES repositories to be imported into a new Repository.
- **Index Creation Tool** (`moses.tools.lib.IndexCreationToolFactory`)
Web based repositories require *.index* files listing objects and directories. This tool creates such files for a file system directory.
- **Dummy Tool** (`moses.tools.lib.DummyToolFactory`)
This tool really does nothing! It can be assigned an icon and used in tool bars as a spacing element.

C Configuring the repository and the tools

The repository and some tools are configured by changing a number of environment variables. These variables themselves are repository objects with an associated type and upon which a number of tools are applicable. The following lists 2 variables that are required to be present for the repository to function correctly. In addition those variables that are present in the default MOSES installation are also described.

C.1 Repository configuration variables

- **DatatypeSystem**

The variable *DatatypeSystem* contains a description of the type system for the repository. Typically when a repository is created a basic data type system will be created that will allow the *DatatypeSystem* variable to be read. Once read the repository's data type system will be extended with the new type information. Hence the type of the *DatatypeSystem* variable must be supported by the repository's basic data type system.

In the standard MOSES release the data type system is represented by a graph written in a special formalism. A compiler is supplied that translates the graph into the format that the file system based repository expects.

- **ToolPoolPath**

In the standard repository user interface a set of repository paths can be defined from which tools are extracted and presented in the tool pane and tool-bars. The *ToolPoolPath* variable contains this set of paths. This variable may be created/modified in two ways: directly by using the *Tools-Path* menu item or by using the *Environment Editor*.

C.2 Moses tools configuration variables

The variables defined in this section use the facilities provided by the *Environment Editor*. This editor stores variables individually as repository objects in the */config/Environment* repository path. These variables are then edited as *Elan* expressions producing either strings or collections of strings in the *Environment Editor*. Any new variables can simply be defined using this editor.

- **SystemConfigurationPath**

Used by the standard repository user interface to locate the base of user interface specific variables.

- **CLASSDIR**

This environment variable is used by the *Model Compiler Tool* to identify the code root directory on the local file system. All generated code files are located with respect to this directory and the component *package* name.

- **ClassPath**

The graph animator tool needs to be able to locate all involved code files. These files may not only be generated from MOSES components but may also include external code. Hence this variable represents a collection of local file system paths and it will typically include the same path defined in the *CLASSDIR* variable.

- **TMP**

This variable identifies a directory into which temporary items can be stored in the file system. An example of a temporary item is the *Java* source code generated from a MOSES component.

- **GraphTypePath**

All MOSES components refer to a formalism description called a *Graph Type*. This graph type is generated on-the-fly from a *Graph Type Description Language* (GTDL) description. The *GraphTypePath* variable contains a collection of repository paths in which such *GTDL* repository objects are assumed to be located.

- **PropertyPath**

Associated with each *Graph Type Description Language* (GTDL) description is a *properties* file that is used to configure the Graph editor. The *PropertyPath* variable contains a collection of repository paths in which such *property* repository objects are assumed to be located.