A Guide to $\mathcal{F}$LORA-*2* Packages

Version 2.0

(Pyrus nivalis)

December 26, 2020

Contents

1 JAVA-to-$\mathcal{F}$LORA-*2* Interfaces **1**

1.1 The Low-level Interface 1

1.2 Debugging $\mathcal{F}$LORA-*2* Statements Used in a Java Program 6

1.2.1 Logging 6

1.2.2 Catching Exceptions, Checking Errors and Warnings 7

1.3 The High-Level Interface (experimental) 8

1.4 Executing Java Application Programs that Call $\mathcal{F}$LORA-*2* 15

1.5 How Do Applications Find the Knowledge Base? 17

1.6 Summary of the Variables and Properties Used by the Interface 18

1.7 Building the Prepackaged Examples 18

2 Qyerying from C/C++ **20**

3 Persistent Modules **21**

3.1 PM Interface 21

3.2 More on DSNs 23

3.3 Examples 24

3.4 Unsupported Features 26

4 SGML and XML Parser for $\mathcal{F}$LORA-*2* **27**

4.1 Introduction 27

4.2 Import Modes for XML in Ergo 28

4.2.1 White Space Handling 28

4.2.2 Requesting Navigation Links 29

4.3 Mapping XML to $\mathcal{F}$LORA-*2* Objects 30

4.3.1 Invention of Object Ids for XML Elements 30

4.3.2 Text and Mixed Element Content 31

4.3.3 Translation of XML Attributes 32
4.3.4 Ordering 33
4.3.5 Additional Attributes and Methods in the `navlinks` Mode 33
4.4 Inspection Predicates 35
4.5 XPath Support 36
4.6 Low-level Predicates 36

Chapter 1

JAVA-to-$\mathcal{F}$LORA-*2* Interfaces

by Aditi Pandit and Michael Kifer

This chapter documents the API for accessing $\mathcal{F}$LORA-*2* from Java programs. The API has two versions: a *low-level API* (used most commonly), which enables Java programs to send arbitrary queries to $\mathcal{F}$LORA-*2* and get results, and an *experimental high-level API*, which is more limited and requires some setup, but can simplify a number of tasks in interfacing the two systems. The high-level API establishes a correspondence between Java classes and $\mathcal{F}$LORA-*2* classes, which enables manipulation of $\mathcal{F}$LORA-*2* classes by executing appropriate methods on the corresponding Java classes. Both interfaces rely on the Java-XSB interface, called *Interprolog* [1], developed by `http://interprolog.com/`.

The API assumes that a Java program is started first and then it invokes XSB/$\mathcal{F}$LORA-*2* as a subprocess. The XSB/$\mathcal{F}$LORA-*2* side is passive: it only responds to the queries sent by the Java side. Queries can be anything that is accepted at the $\mathcal{F}$LORA-*2* shell prompt: queries, insert/delete commands, control switches, etc., are all fine. One thing to remember is that the backslash is used in Java as an escape symbol and in $\mathcal{F}$LORA-*2* as a prefix of the builtin operators and commands. Therefore, each backslash must be escaped with another backslash. That is, instead of a query like "`p(?X) \and q(?X).`" the API requires "`p(?X) \\and q(?X).`".

The `FloraObject` object. While reading this document one will notice that the class `FloraObject` is used in many cases. This class consists of Java objects that encapsulate $\mathcal{F}$LORA-*2* objects. These Java objects are mostly used internally. From the end user's point of view, the only method of interest in this class is `toString()`.

1.1 The Low-level Interface

The low-level API enables Java programs to send arbitrary queries to $\mathcal{F}$LORA-*2* and get results. It is assumed that the following two Java properties are set either as part of the java command (e.g., `java ... -DPROLOGDIR=some-dir ...`) or inside the Java application itself, e.g.,

```
System.setProperty("PROLOGDIR","C:\\JSmith\\XSB\\config\\x64-pc-windows\\bin");
```

Please remember that Windows uses backslash as a file separator, and inside a Java program these backslashes must be doubled, as shown above.

The two aforementioned properties are:

`PROLOGDIR:` This variable points to the folder containing the XSB executable (binary, not the command script).
To get the right value for your installation, start $\mathcal{F}$LORA-*2* and execute this at the prompt:

```
system{bindir = ?D}.
```

The result will be returned in the variable `?D`.

`FLORADIR:` This variable must point to the folder containing the $\mathcal{F}$LORA-*2* installation.
To get the right value for your installation, start $\mathcal{F}$LORA-*2* and execute this at the prompt:

```
system{installdir = ?D}.
```

Again, the result will be returned in the variable `?D`.

In order to be able to access $\mathcal{F}$LORA-*2*, the Java program must first establish a session for a running instance of $\mathcal{F}$LORA-*2*. Multiple sessions can be active at the same time. The knowledge bases in the different running instances are completely independent. Sessions are instances of the class `net.sf.flora2.API.FloraSession`. This class provides methods for opening/closing sessions and loading $\mathcal{F}$LORA-*2* knowledge bases (which are also used in the high-level interface). In addition, a session provides methods for executing arbitrary $\mathcal{F}$LORA-*2* queries. The following is the complete list of the methods that are available in that class. All these are *public instance* methods and the word "public" is therefore omitted.

- `FloraSession()`

  This constructor creates a connection to an instance of $\mathcal{F}$LORA-*2*. Use it like this:

  ```
  FloraSession session = new FloraSession();
  ```

  All the methods below are executed on `FloraSession`-objects produced in this way.

- `close()`
  This method must be called to terminate a $\mathcal{F}$LORA-*2* session. Note that this does not terminate the Java program that initiated the session: to exit the Java program that talks to $\mathcal{F}$LORA-*2*, one needs to execute the statement

  ```
  System.exit();
  ```

  Note that just returning from the `main` method is not enough.

- `Iterator<FloraObject> executeQuery(String query)`

  This method executes the $\mathcal{F}$LORA-*2* query given by the parameter `query`. The query must be terminated with a period, exactly as it would be typed in the $\mathcal{F}$LORA-*2* shell. It is used to execute $\mathcal{F}$LORA-*2* queries that do not require variable bindings to be returned back to Java *or* queries that have only a single variable to be returned. Each binding is represented as an instance of the class `net.sf.flora2.API.FloraSession`. The examples below illustrate how to process the results returned by this method.

- `Iterator<HashMap<String,FloraObject>> executeQuery(String query,Vector vars)`

  This method executes the $\mathcal{F}$LORA-*2* query given by the first argument. The query must be terminated with a period, as if it were typed in the $\mathcal{F}$LORA-*2* shell. The Vector `vars` (of strings) specifies the names of all the variables in the query for which bindings need to be returned. These variables are added to the vector using the method `add` before calling `executeQuery`. For instance, `vars.add("?X")`.

  This version of `executeQuery` returns an iterator over all bindings returned by the $\mathcal{F}$LORA-*2* query. Each binding is represented by a `HashMap<String,FloraObject>` object which can be used to obtain the value of each variable in the query (using the `get()` method). The value of each variable returned is an instance of `net.sf.flora2.API.FloraObject`.

  The examples below show how to handle the results returned by this method.

- `boolean executeCommand(String command)`
  This is a simplified way of executing $\mathcal{F}$LORA-*2* queries that *do not need* to return any results, i.e., when the user wants to know if the query is true or false for *some* bindings of command's arguments (if there are any), but not the actual bindings. There are also differences (compared to `executeQuery()`) in the way this command handles exceptions, as explained in the next section. As before, the command must be terminated with a period.

- `boolean loadFile(String fileName,String moduleName)`

  This method loads the $\mathcal{F}$LORA-*2* program, specified by the parameter `fileName` into the $\mathcal{F}$LORA-*2* module specified in `moduleName`. If errors occur during loading, `loadFile()` returns `false`.

- `boolean compileFile(String fileName,String moduleName)`

  This method compiles (but does not load) the $\mathcal{F}$LORA-*2* program, specified by the parameter `fileName` for the $\mathcal{F}$LORA-*2* module specified in `moduleName`. If errors occur during compilation, `compileFile()` returns `false`.

- `boolean addFile(String fileName,String moduleName)`

  This method adds the $\mathcal{F}$LORA-*2* program, specified by the parameter `fileName` to an existing $\mathcal{F}$LORA-*2* module specified in `moduleName`. If errors occur during addition, `addFile()` returns `false`.

- `boolean compileaddFile(String fileName,String moduleName)`

This method compiles the $\mathcal{F}$LORA-*2* program, specified by the parameter `fileName` for addition to the $\mathcal{F}$LORA-*2* module specified in `moduleName`. If errors occur during compilation, `compileaddFile()` returns `false`.

The code snippet below illustrates the low-level API.

Step 1: Writing $\mathcal{F}$LORA-*2* programs to be called by Java. Let us assume that we have a file, called `flogic_basics.flr`, which contains the following information:

```
person :: object.
dangerous_hobby :: object.
john:employee.
employee::person.

bob:person.
tim:person.
betty:employee.

person[|age=>integer,
       kids=>person,
       salary(year)=>value,
       hobbies=>hobby,
       believes_in=>something,
       instances => person
|].

mary:employee[
    age->29,
    kids -> {tim,leo,betty},
    salary(1998) -> a_lot
].

tim[hobbies -> {stamps, snowboard}].
betty[hobbies->{fishing,diving}].

snowboard:dangerous_hobby.
diving:dangerous_hobby.

?_X[self-> ?_X].

person[|believes_in -> {something, something_else}|].
```

Step 2: Writing a JAVA application to interface with $\mathcal{F}$LORA-*2*. The following code loads a $\mathcal{F}$LORA-*2* program from a file and then passes queries to the knowledge base.

```
import java.util.*;
```

```
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class flogicbasicsExample {

    public static void main(String[] args) {
        // create a new session for a running instance of the engine
        FloraSession session = new FloraSession();
        System.out.println("Engine session started");

        // Assume that Java was called with -DINPUT_FILE=the-file-name
        String fileName = System.getProperty("INPUT_FILE");
        if(fileName == null || fileName.trim().length() == 0) {
            System.out.println("Invalid path to example file!");
            System.exit(0);

        }
        // load the program into module basic_mod
        if (session.loadFile(fileName,"basic_mod"))
            System.out.println("Example loaded successfully!");
        else
            System.out.println("Error loading the example!");

        /* Running queries from flogic_basics.flr */

        /* Query for persons */
        String command = "?X:person@basic_mod.";
        System.out.println("Query:"+command);
        Iterator<FloraObject> personObjs = session.executeQuery(command);

        /* Printing out the person names and information about their kids */
        while (personObjs.hasNext()) {
            FloraObject personObj = personObjs.next();
            System.out.println("Person name:"+personObj);
        }

        command = "person[instances -> ?X]@basic_mod.";
        System.out.println("Query:"+command);
        personObjs = session.executeQuery(command);

        /* Printing out the person names  */
        while (personObjs.hasNext()) {
            Object personObj = personObjs.next();
            System.out.println("Person Id: "+personObj);
        }

        /* Example of executeQuery with two arguments */
```

```
        Vector<String> vars = new Vector<String>();
        vars.add("?X");
        vars.add("?Y");

        Iterator<HashMap<String,FloraObject>> allmatches =
            session.executeQuery("?X[believes_in -> ?Y]@basic_mod.",vars);
        System.out.println("Query:?X[believes_in -> ?Y]@basic_mod.");
        while(allmatches.hasNext()) {
            HashMap<String,FloraObject> firstmatch = allmatches.next();
            Object Xobj = firstmatch.get("?X");
            Object Yobj = firstmatch.get("?Y");
            System.out.println(Xobj+" believes in: "+?Yobj);
        }
        // quit the system
        session.close();
        System.exit(0);
    }
}
```

For the information on how to invoke the above Java class in the context of the Java-$\mathcal{F}$LORA-*2* API, please see Section 1.4.

1.2 Debugging $\mathcal{F}$LORA-*2* Statements Used in a Java Program

1.2.1 Logging

It often happens that an $\mathcal{F}$LORA-*2* query or a file used from within a Java Program has an error and `executeQuery()` returns an error message saying that there is a "problem" with a $\mathcal{F}$LORA-*2* statement. The Java part does not know what the problem is but it can be told to show the output of the $\mathcal{F}$LORA-*2* statements on the console. This can be done by executing the statement

```
FloraSession.showOutput();
```

This will stream all warnings, errors, and just normal output from $\mathcal{F}$LORA-*2* to the console. When no longer needed, this mode can be turned off like this:

```
FloraSession.hideOutput();
```

The `FloraSession.showOutput();` statement can be executed at any point in the program, but preferably after calling `FloraSession()`, to avoid irrelevant output.

Here is an example of what you might see:

```
My command 2 has succeeded
```

```
yes

Loading file test.ergo:

yes

++Error[Ergo]> [test.ergo] <Composer> near line(2)/char(3) 'b' unexpected operand:
 ',', '.', or some other operator may be missing just before the indicated location

++1 error

++compilation aborted
```

Here the output from $\mathcal{F}$LORA-*2* starts with normal output from, say, `writeln(...)@\io` commands and ends with a compilation error encountered while compiling the file `test.ergo`. (The above is output from $\mathcal{E}$RGO; in $\mathcal{F}$LORA-*2* it will be similar.)

Two other useful calls are

```
FloraSession.enableLogging();
FloraSession.disableLogging();
```

Executing `FloraSession.enableLogging();` will cause the Java API to record all major events such as starting a session, ending it, loading or adding a $\mathcal{F}$LORA-*2* file, and execution of every query. It does not report query answers, however. Logging can be enabled or disabled anywhere in the program, but, of course, the log messages will start to appear only after the enabling command is executed and will cease to appear after `disableLogging()` is executed.

1.2.2 Catching Exceptions, Checking Errors and Warnings

Logging allows one to find errors and warnings in an $\mathcal{F}$LORA-*2* subprocess of a Java program by checking the output produced by the session. However, often one needs to be able to do this *programmatically* and this can be done as follows.

Exceptions. First, if a running $\mathcal{F}$LORA-*2* query invoked via `executeQuery()` issues a run-time error, a Java `FlrException` is thrown, and this can be caught by the parent Java program. An exception is also thrown if the query passed to `executeQuery()` has a syntax error.[1] Note that $\mathcal{F}$LORA-*2* statements invoked via the other methods (`executeCommand()`, `loadFile()`, etc.) do *not* throw exceptions and errors that occur during the execution of those statements can be detected only through the mechanism of `hasErrors()` described below. However, `executeCommand()`, `loadFile()`, `addFile()`, etc., may throw exceptions for other reasons, such as a wrong module in `loadFule()`, etc.

[1] If the query statement itself has no syntax error but it loads a file that has a syntax error then *no* exception is thrown. Read on to see how to identify this situation programmatically.

Detecting syntax errors and warnings. Errors and warnings produced by the *FLORA-2* compiler do *not* result in exceptions, so they cannot be caught via Java's try-catch mechanism. To detect if a previous `executeQuery()`, `executeCommand()`, `loadFile()`, or similar API command produced a warning or an error, use the following *public instance* methods in class `FloraSession`:

- `boolean hasErrors()` — executing `session.hasErrors()`, where session is a variable holding a `FloraSession` object created earlier, will tell if a previous `executeQuery()` or other such command (executed on the same `FloraSession` object) has produced an error. This also includes runtime errors, like `FlrException`'s.
- `boolean hasWarnings()` — executing `session.hasWarnings()`, where session is a variable holding a `FloraSession` object, will tell if a previous `executeQuery()` or other such command (executed on the same `FloraSession` object) has produced a warning.

1.3 The High-Level Interface (experimental)

The high-level API operates by creating proxy Java classes for *FLORA-2* classes selected by the user. This enables the Java program to operate on *FLORA-2* classes by executing appropriate methods on the corresponding proxy Java classes. However, compared to the low-level interface, the high-level one is somewhat limited. Both interfaces can be used at the same time, if desired.

Note A: Most users appear to opt for the low-level interface and such readers can skip this section.

Note B: This interface will not work for *FLORA-2* programs that use *non-alphanumeric* names for methods and predicates. For instance, if a program involves statements like `foo['bar$#123'->456]` then the interface might generate syntactically incorrect Java proxy classes and errors will be issued during the compilation.

The use of the high-level API involves a number of steps, as described below.

Stage 1: Writing *FLORA-2* programs to use with the high-level interface. We assume the same `flogic_basics.flr` file as in the previous example.

Stage 2: Generating Java classes that serve as proxies for *FLORA-2* classes. The *FLORA-2* side of the Java-to-*FLORA-2* high level API provides a predicate to generate Java proxy classes for each F-logic class which have a signature declaration in the *FLORA-2* knowledge base. A proxy class gets defined so that it would have methods to manipulate the attributes and methods of the corresponding F-logic class for which signature declarations are available. If an F-logic class has a declared value-returning attribute `foobar` then the proxy class will have the following methods. Each method name has the form $action S_1 S_2 S_3$_`foobar`, where *action* is either `get`, `set`, or `delete`. The specifier S_1 indicates the type of the method — `V` for value-returning, `B` for Boolean, and `P` for procedural. The specifier S_2 tells whether the operation applies to the signature of the method (`S`), e.g., `person[foobar=>string]`, or

to the actual data (`D`), for example, `john[foobar->3]`. Finally, the specifier S_3 tells if the operation applies to the inheritable variant of the method (`I`) or its non-inheritable variant (`N`).

1. `public Iterator<FloraObject> getVDI_foobar()`
   `public Iterator<FloraObject> getVDN_foobar()`
   `public Iterator<FloraObject> getVSI_foobar()`
   `public Iterator<FloraObject> getVSN_foobar()`
   The above methods query the knowledge base and get all answers for the attribute `foobar`. They return iterators through which these answers can be processed one-by-one. Each object returned by the iterator is of type `FloraObject`. The `getVDN` form queries non-inheritable data methods and `getVDI` the inheritable ones. The `getVSI` and `getVSN` forms query the signatures of the attribute `foobar`.

2. `public boolean setVDI_foobar(Vector value)`
   `public boolean setVDN_foobar(Vector value)`
   `public boolean setVSI_foobar(Vector value)`
   `public boolean setVSN_foobar(Vector value)`
   These methods add values to the set of values returned by the attribute `foobar`. The values must be placed in the vector parameter passed these methods. Again, `setVDN` adds data for non-inheritable methods and `setVDI` is used for inheritable methods. `setVSI` and `setVSN` add types to signatures.

3. `public boolean setVDI_foobar(Object value)`
   `public boolean setVDN_foobar(Object value)`
   `public boolean setVSI_foobar(Object value)`
   `public boolean setVSN_foobar(Object value)`
   These methods provide a simplified interface when only one value needs to be added. It works like the earlier set_* methods, except that only one value given as an argument is added.

4. `public boolean deleteVDI_foobar(Vector value)`
   `public boolean deleteVDN_foobar(Vector value)`
   `public boolean deleteVSI_foobar(Vector value)`
   `public boolean deleteVSN_foobar(Vector value)`
   Delete a set of values of the attribute `foobar`. The set is specified in the vector argument.

5. `public boolean deleteVDI_foobar(Object value)`
   `public boolean deleteVDN_foobar(Object value)`
   `public boolean deleteVSI_foobar(Object value)`
   `public boolean deleteVSN_foobar(Object value)`
   A simplified interface for the case when only one value needs to be deleted.

6. `public boolean deleteVDI_foobar()`
   `public boolean deleteVDN_foobar()`
   `public boolean deleteVSI_foobar()`
   `public boolean deleteVSN_foobar()`
   Delete all values for the attribute `foobar`.

For F-logic methods with arguments, the high-level API provides Java methods as above, but they take more arguments to accommodate the parameters that F-logic methods take. Let us assume that the F-logic method is called `foobar2` and it takes parameters `arg1` and `arg2`. As before the `getVDI_*`, `setVDI_*`, etc., forms of the Java methods are for dealing with inheritable $\mathcal{F}$LORA-*2* methods and the `getVDN_*`, `setVDN_*`, etc., forms are for dealing with non-inheritable $\mathcal{F}$LORA-*2* methods.

1. `public Iterator<FloraObject> getVDI_foobar2(Object arg1, Object arg2)`
   `public Iterator<FloraObject> getVDN_foobar2(Object arg1, Object arg2)`
   Obtain all values for the F-logic method invocation `foobar2(arg1,arg2)`.

2. `public boolean setVDI_foobar2(Object arg1, Object arg2, Vector value)`
   `public boolean setVDN_foobar2(Object arg1, Object arg2, Vector value)`
   Add a set of methods specified in the parameter `value` for the method invocation `foobar2(arg1,arg2)`.

3. `public boolean setVDI_foobar2(Object arg1, Object arg2, Object value)`
   `public boolean setVDN_foobar2(Object arg1, Object arg2, Object value)`
   A simplified interface when only one value is to be added.

4. `public boolean deleteVDI_foobar2(Object arg1, Object arg2, Vector value)`
   `public boolean deleteVDN_foobar2(Object arg1, Object arg2, Vector value)`
   Delete a set of values from `foobar2(arg1,arg2)`. The set is given by the vector parameter `value`.

5. `public boolean deleteVDI_foobar2(Object arg1, Object arg2, Object value)`
   `public boolean deleteVDN_foobar2(Object arg1, Object arg2, Object value)`
   A simplified interface for deleting a single value.

6. `public boolean deleteVDI_foobar2(Object arg1, Object arg2)`
   `public boolean deleteVDN_foobar2(Object arg1, Object arg2)`
   Delete all values for the method invocation `foobar2(arg1,arg2)`.

For Boolean and procedural methods, the generated methods are similar except that there is only one version for the set and delete methods. In addition, Boolean inheritable methods use the `getBDI_*`, `setBDI_*`, etc., form, while non-inheritable methods use the `getBDN_*`, etc., form. Procedural methods use the `getPDI_*`, `getPDN_*`, etc., forms. For instance,

1. `public boolean getBDI_foobar3()`
   `public boolean getBDN_foobar3()`
   `public boolean getPDI_foobar3()`
   `public boolean getPDN_foobar3()`

2. `public boolean setBDI_foobar3()`
   `public boolean setBDN_foobar3()`
   `public boolean setPDI_foobar3()`
   `public boolean setPDN_foobar3()`

3. `public boolean deleteBDI_foobar3()`
   `public boolean deleteBDN_foobar3()`
   `public boolean deletePDI_foobar3()`
   `public boolean deletePDN_foobar3()`

In addition, the methods to query the ISA hierarchy are available:

- `public Iterator<FloraObject> getDirectInstances()`
- `public Iterator<FloraObject> getInstances()`
- `public Iterator<FloraObject> getDirectSubClasses()`
- `public Iterator<FloraObject> getSubClasses()`
- `public Iterator<FloraObject> getSuperClasses()`
- `public Iterator<FloraObject> getDirectSuperClasses()`

These methods apply to the java proxy object that corresponds to the F-logic class person.

All these methods are generated automatically by executing the following $\mathcal{F}$LORA-*2* query (defined in the `javaAPI` package in $\mathcal{F}$LORA-*2*). All arguments in the query must be bound:

```
// write(?Class,?Module,?ProxyClassFileName).
?- write(foo,example,'myproject/foo.java').
```

The first argument specifies the class for which to generate the methods, the file name tells where to put the Java file for the proxy object, and the model argument tells which $\mathcal{F}$LORA-*2* model to load this program to. The result of this execution will be the file `foo.java` which should be included with your java program (the program that is going to interface with $\mathcal{F}$LORA-*2*). Note that because of the Java conventions, the file name must have the same name as the class name. It is important to remember, however, that proxy methods will be generated only for those F-logic methods that have been declared using signatures.

Let us now come back to our program `flogic_basics.flr` for which we want to use the high-level API. Suppose we want to query the person class. To generate the proxy declarations for that class, we create the file `person.java` for the module `basic_mod` as follows.

```
?- load{'examples/flogic_basics'>>basic_mod}.
?- load{javaAPI}.
?- write(person,basic_mod,'examples/person.java')@\prolog
```

The `write` method will create the file `person.java` shown below. The methods defined in `person.java` are the class constructors for `person`, the methods to query the ISA hierarchy, and the "get", "set" and "delete" methods for each method and attribute declared in the $\mathcal{F}$LORA-*2* class `person`. The parameters for the "get", "set" and "delete" Java methods are the same as for the corresponding $\mathcal{F}$LORA-*2* methods. The first constructor for class `person` takes a low-level object of class `net.sf.flora2.API.FloraObject` as a parameter. The second parameter is the $\mathcal{F}$LORA-*2* module for which the proxy object is to be created.

The second `person`-constructor takes F-logic object Id instead of a low-level `FloraObject`. It also takes the module name, as before, but, in addition, it takes a session for a running $\mathcal{F}$LORA-*2* instance. The session parameter was not needed for the first `person`-constructor because `FloraObject` is already attached to a concrete session.

It can be seen from the form of the proxy object constructors that proxy objects are attached to specific $\mathcal{F}$LORA-*2* modules, which may seem to go against the general philosophy that F-logic objects do not belong to any module — only their methods do. On closer examination, however, attaching high-level proxy Java objects to modules makes perfect sense. Indeed, a proxy object encapsulates operations for manipulating F-logic attributes and methods, which belong to concrete $\mathcal{F}$LORA-*2* modules, so the proxy object needs to know which module it operates upon.

person.java file

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class person {

  public FloraObject sourceFloraObject;

  // proxy objects' constructors
  public person(FloraObject sourceFloraObject, String moduleName){ ... }
  public person(String floraOID,String moduleName, FloraSession session){...}

  // ISA hierarchy queries
  public Iterator<FloraObject> getDirectInstances() { ... }
  public Iterator<FloraObject> getInstances() { ... }
  public Iterator<FloraObject> getDirectSubClasses() { ... }
  public Iterator<FloraObject> getSubClasses() { ... }
  public Iterator<FloraObject> getDirectSuperClasses() { ... }
  public Iterator<FloraObject> getSuperClasses() { ... }

  // Java methods for manipulating methods
  public boolean setVDI_age(Object value) { ... }
  public boolean setVDN_age(Object value) { ... }
  public Iterator<FloraObject> getVDI_age(){ ... }
  public Iterator<FloraObject> getVDN_age(){ ... }
  public boolean deleteVDI_age(Object value) { ... }
  public boolean deleteVDN_age(Object value) { ... }
  public boolean deleteVDI_age() { ... }
  public boolean deleteVDN_age() { ... }
  public boolean setVDI_salary(Object year,Object value) { ... }
  public boolean setVDN_salary(Object year,Object value) { ... }
  public Iterator<FloraObject> getVDI_salary(Object year) { ... }
  public Iterator<FloraObject> getVDN_salary(Object year) { ... }
```

```
  public boolean deleteVDI_salary(Object year,Object value) { ... }
  public boolean deleteVDN_salary(Object year,Object value) { ... }
  public boolean deleteVDI_salary(Object year) { ... }
  public boolean deleteVDN_salary(Object year) { ... }
  public boolean setVDI_hobbies(Vector value) { ... }
  public boolean setVDN_hobbies(Vector value) { ... }
  public Iterator<FloraObject> getVDI_hobbies(){ ... }
  public Iterator<FloraObject> getVDN_hobbies(){ ... }
  public boolean deleteVDI_hobbies(Vector value) { ... }
  public boolean deleteVDN_hobbies(Vector value) { ... }
  public boolean deleteVDI_hobbies(){ ... }
  public boolean deleteVDN_hobbies(){ ... }
  public boolean setVDI_instances(Vector value) { ... }
  public boolean setVDN_instances(Vector value) { ... }
  public Iterator<FloraObject> getVDI_instances(){ ... }
  public Iterator<FloraObject> getVDN_instances(){ ... }
  public boolean deleteVDI_instances(Vector value) { ... }
  public boolean deleteVDN_instances(Vector value) { ... }
  public boolean deleteVDI_instances(){ ... }
  public boolean deleteVDN_instances(){ ... }
  public boolean setVDI_kids(Vector value) { ... }
  public boolean setVDN_kids(Vector value) { ... }
  public Iterator<FloraObject> getVDI_kids(){ ... }
  public Iterator<FloraObject> getVDN_kids(){ ... }
  public boolean deleteVDI_kids(Vector value) { ... }
  public boolean deleteVDN_kids(Vector value) { ... }
  public boolean deleteVDI_kids(){ ... }
  public boolean deleteVDN_kids(){ ... }
  public boolean setVDI_believes_in(Vector value) { ... }
  public boolean setVDN_believes_in(Vector value) { ... }
  public Iterator<FloraObject> getVDI_believes_in(){ ... }
  public Iterator<FloraObject> getVDN_believes_in(){ ... }
  public boolean deleteVDI_believes_in(Vector value) { ... }
  public boolean deleteVDN_believes_in(Vector value) { ... }
  public boolean deleteVDI_believes_in(){ ... }
  public boolean deleteVDN_believes_in(){ ... }
}
```

Stage 3: Writing Java applications that use the high-level API. The following program (`flogicbasicsExample.java`) shows several queries that use the high-level interface. The class `person.java` is generated at the previous stage. The methods of the high-level interface operate on Java objects that are proxies for $\mathcal{F}$LORA-*2* objects. These Java objects are members of the class `net.sf.flora2.API.FloraObject`. Therefore, before one can use the high-level methods one need to first retrieve the appropriate proxy objects on which to operate. This is done by sending an appropriate query through the method `executeQuery`—the same method that was used in the low-level interface. Alternatively, `person`-objects could

be constructed using the 3-argument proxy constructor, which takes F-logic oids.

```
import java.util.*;
import net.sf.flora2.API.*;
import net.sf.flora2.API.util.*;

public class flogicbasicsExample {

   public static void main(String[] args) {
     /* Initializing the session */
     FloraSession session = new FloraSession();
     System.out.println("Flora session started");

     String fileName = "examples/flogic_basics"; // must be a valid path
     /* Loading the flora file */
     if (session.loadFile(fileName,"basic_mod"))
         System.out.println("Example loaded successfully!");
     else
         System.out.println("Error loading the example!");

     // Retrieving instances of the class person through low-level API
     String command = "?X:person@basic_mod.";
     System.out.println("Query:"+command);
     Iterator<FloraObject> personObjs = session.executeQuery(command);

     /* Print out person names and information about their kids */
     person currPerson = null;
     while (personObjs.hasNext()) {
         FloraObject personObj = personObjs.next();
         // Elevate personObj to the higher-level person-object
         currPerson =new person(personObj,"basic_mod");

         /* Set that person's age to 50 */
         currPerson.setVDN_age("50");

         /* Get this person's kids */
         Iterator<FloraObject> kidsItr = currPerson.getVDN_kids();
         while (kidsItr.hasNext()) {
             FloraObject kidObj = kidsItr.next();
             System.out.println("Person: " + personObj + " has kid: " +kidObj);

             person kidPerson = null;
             // Elevate kidObj to kidPerson
             kidPerson = new person(kidObj,"basic_mod");

             /* Get kidPerson's hobbies */
             Iterator<FloraObject> hobbiesItr = kidPerson.getVDN_hobbies();
```

```
            while(hobbiesItr.hasNext()) {
                FloraObject hobbyObj = hobbiesItr.next();
                System.out.println("Kid:"+kidObj + " has hobby:" +hobbyObj);
            }
        }
    }

    FloraObject age;
    // create a person-object directly by supplying its F-logic OID
    // father(mary)
    currPerson = new person("father(mary)", "example", session);
    Iterator<FloraObject> maryfatherItr = currPerson.getVDN_age();
    age = maryfatherItr.next();
    System.out.println("Mary's father is " + age + " years old");

    // create a proxy object for the F-logic class person itself
    person personClass = new person("person", "example", session);
    // query its instances through the high-level interface
    Iterator<FloraObject> instanceIter = personClass.getInstances();
    System.out.println("Person instances using high-level API:");
    while (instanceIter.hasNext())
        System.out.println("    " + instanceIter.next());

    session.close();
    System.exit();
  }
}
```

1.4 Executing Java Application Programs that Call $\mathcal{F}$LORA-*2*

To compile and run Java programs that interface with $\mathcal{F}$LORA-*2*, follow the following guidelines.

- *Compilation*: Place the files `flogicsbasicsExample.java` (the program you have written) and `person.java` (the automatically generated file) in the same directory and compile them using the `javac` command. Add the jar-files containing the API code and `interprolog.jar` to the classpath using the `-classpath` parameter (the first line is for Windows and the second for Mac and Linux):

  ```
  -classpath "%FLORADIR%\java\flora2java.jar";"%FLORADIR%\java\interprolog.jar"
  -classpath "$FLORADIR/java/flora2java.jar":"$FLORADIR/java/interprolog.jar"
  ```

  `FLORADIR` here is a shell (cmd, in Windows) variable that can be set by the scripts `flora_settings.sh` (Linux/Mac) or `flora_settings.bat` (Windows). In sum, the Java compilation command should look as below (again, the first command below is for Windows and the second for Linux/Mac):

```
path-to\javac -classpath
     "%FLORADIR%\java\flora2java.jar";"%FLORADIR%\java\interprolog.jar"
path-to/javac -classpath
     "$FLORADIR/java/flora2java.jar":"$FLORADIR/java/interprolog.jar"
```

Note: each of the above commands should be on one line.

- *Running*: Generally, Java programs that call $\mathcal{F}$LORA-*2* should be invoked using the following command. For Linux and Mac, change %*VAR*% to $*VAR*:

```
path-to\java -DPROLOGDIR=%PROLOGDIR%
                -DFLORADIR=%FLORADIR%
                -Djava.library.path=%PROLOGDIR%            <--- optional
                -classpath %MYCLASSPATH% flogicbasicsExample
```

  The above commands use several shell/cmd variables that are explained below. Instead of using the variables, one can substitute their values directly—read on.

  - If the `javac` and `java` commands can be found through the `PATH` environment variable then one can simply type `javac` and `java` instead of the above. On Linux and Mac this is almost always the case, but on Windows one might have to set the `PATH` variable explicitly.
  - `PROLOGDIR`: This variable should point to the directory containing the XSB executable, which can be accomplished by executing the scripts `flora2\java\flora_settings.bat` (Windows) or `flora2/java/flora_settings.sh` (Linux/Mac).
    Alternatively, one can type `-DPROLOGDIR=`*path-to-prolog-bindir* in the above `java` command, where the value to substitute for *path-to-prolog-bindir* can be obtained by executing this query at the prompt:

    ```
    system{bindir = ?D}.
    ```

    The result will be returned as a binding for the variable `?D`.
  - `FLORADIR`: This variable should be set to the directory containing the $\mathcal{F}$LORA-*2* system, which can be done by executing the aforesaid scripts `flora_settings.bat` and `flora_settings.sh`.
    Alternatively, one can type `-DFLORADIR=`*path-to-flora-dir* in the above `java` command, where the value to substitute for *path-to-flora-dir* can be obtained by executing this query at the prompt:

    ```
    system{installdir = ?D}.
    ```

    Again, the result will be returned as a binding of the variable `?D`.
  - `MYCLASSPATH`: This variable should include the correct paths to the jar files containing the API code, i.e., `flora2java.jar` and file `interprolog.jar`, plus the directory where the main application class (like `flogicbasicsExample` in our example) is found. Normally, one sets `MYCLASSPATH` to `%CLASSPATH%;%FLORADIR%\java\flora2java.jar;%FLORADIR%\java\interprolog.jar;`

`DirOfTheExample`, where `DirOfTheExample` is the directory where the main application class resides. In our example, this directory is simply . (the current directory). For Linux and Mac, use ':' instead of ';' as a separator, forward slashes instead of backward ones, and *$VAR* instead of *%VAR%*.

One can, of course, substitute the contents of the `MYCLASSPATH` variable directly into the `-classpath %MYCLASSPATH%` part of the above Java/Javac invocation commands.

– The variable `java.library.path` in the above command is optional. It needs to be set *only if XSB is configured to use the native Java interface* (which usually is not the case).

- Some Java applications may employ additional Java properties. For instance, the program that uses the low-level API in Section 1.1 (in Step 2) has the line

```
String fileName = System.getProperty("INPUT_FILE");
```

which means that it expects the property `INPUT_FILE` to be set with the `-D` option at the Java invocation time. In general, such additional properties can be also set via the method `System.setProperty()` inside the Java application. In our particular case, the program expects that `INPUT_FILE` is set to point to the `flogic_basics.flr` FLORA-*2* file, which it then loads. In other words, the `java` command shown above also needs this parameter:

```
-DINPUT_FILE="%INPUT_FILE%"    (Windows)
-DINPUT_FILE="$INPUT_FILE"     (Linux/Mac)
```

In general, one such additional parameter is needed for each property that the Java application queries using the `getProperty()` method.

1.5 How Do Applications Find the Knowledge Base?

When a Java application starts FLORA-*2*, the latter determines the default runtime directory in which it will work. Usually, this is the directory in which your Java application runs. You can find out which directory it is by sending the following query to FLORA-*2*:

```
File[cwd->?Dir]@\io.
```

`?Dir` will be bound to the runtime directory and Java can get that value as explained earlier. Your Java application can change that directory via this query:

```
File[chdir('....new current dir...')]@\io.
```

The simplest basic rule is that all FLORA-*2*'s files that your Java application loads, adds, etc., must be specified relative to the current directory.

One can also put additional directories to the FLORA-*2*'s search path by executing the query

```
Libpath[add('....new dir to search...')]@\sys.
```

Then your application can use file names not only relative to the runtime directory but also relative to any of the directories added in this way. Note that this may put many directories on the search path, and several of them may have similarly named files. Therefore, one must make sure that the search is unambiguous.

1.6 Summary of the Variables and Properties Used by the Interface

The Java-$\mathcal{F}$LORA-*2* interface needs the following variables and properties to be set:

- `JAVA_HOME` – this is an OS environment variable. It is normally set when you install Java. Normally, Java will not work correctly if this environment variable is not set correctly.
- The following Java properties must be set for the Java API to work. They can be set either through the `-D` option of the `java` command or inside the Java application via `System.setProperty("propertyname",value)`.
  - `FLORADIR` — the path to the $\mathcal{F}$LORA-*2* installation directory.
  - `PROLOGDIR` — the path to the folder containing XSB executable.

  The proper values for these properties can be obtained from $\mathcal{F}$LORA-*2* by running these queries, respectively:

  ```
  system{installdir=?D}.
  system{bindir=?D}.
  ```

- The following shell/cmd variable is useful, if you do not know where the `java` and `javac` executables are on your machine. This may be needed on Windows, if your JDK installer failed to set the Windows `PATH` environment variable so that the system would find these commands easily. (On Linux and Mac the above commands are usually fund through the `PATH` variable.)
  - `JAVA_BIN` — the directory where Java executables `java` and `javac` live. It is usually set to `$JAVA_HOME/bin` or `%JAVA_HOME%\bin`, depending on the OS.

  This variable can be set by `unixVariables.sh` or `windowsVariables.bat`, whichever applies to your OS.

1.7 Building the Prepackaged Examples

Sample applications of the Java-$\mathcal{F}$LORA-*2* interface are found in the `java/API/examples` folder of the $\mathcal{F}$LORA-*2* distribution. To build the examples, use the scripts `buildExample.sh` or `buildExample.bat` in the `java/API/examples` folder, whichever applies. For instance, to build the `flogicbasicsExample` example, use these commands on Linux, Mac, and other Unix-like systems:

```
cd examples
buildExample.sh flogicbasicsExample
```

On Windows, use this:

```
cd examples
buildExample.bat flogicbasicsExample
```

To run the demos, use the scripts `runExample.sh` or `runExample.bat` in `java/API/examples`. For instance, to run the `flogicbasicsExample`, use this command on Linux and Mac:

```
runExample.sh flogicbasicsExample
```

On Windows, use this:

```
runExample.bat flogicbasicsExample
```

Chapter 2

Querying $\mathcal{F}$LORA-*2* from C/ C++ Programs

by Michael Kifer

This API enables one to start a $\mathcal{F}$LORA-*2* session from within a C or C++ program, load knowledge bases, and query them via that API. The details appear in the document "*About Querying Ergo from C and C++*" in the $\mathcal{E}$RGO$\mathcal{AI}$ *Example Bank* `https://drive.google.com/open?id=1bfoj-yegVdpBnJ6BfgKqKW6p-qmlhfkwU2p0oBYf2oU`. A well-documented running example is also provided in that example bank: `https://drive.google.com/open?id=1Mux-wkYRXwCV1B9ZDOO01xwFxEGzGnYq`. Although the document and the program refer to $\mathcal{E}$RGO, they equally apply to $\mathcal{F}$LORA-*2*, if `ergo_query()` there is replaced with `flora_query()`. The main issues one should to pay attention to are:

- Compilation of C/C++ programs that invoke $\mathcal{F}$LORA-*2*. The process is different for Windows, Linux, and Mac.
- Writing C/C++ programs that interface with $\mathcal{F}$LORA-*2*.

The last aspect has four major steps:

- Initialization of XSB.
- Initialization of $\mathcal{F}$LORA-*2*.
- Issuing queries to $\mathcal{F}$LORA-*2* and getting the results.
- Error handling.

All of these issues are detailed in the aforesaid document and the accompanying sample program.

Chapter 3

Persistent Modules

by Vishal Chowdhary

This chapter describes a $\mathcal{F}$LORA-*2* package that enables persistent modules. A *persistent module* (abbr., PM) is like any other $\mathcal{F}$LORA-*2* module except that it is associated with a database. Any insertion or deletion of base facts and rules in such a module results in a corresponding operation on the associated database. This data persists across $\mathcal{F}$LORA-*2* sessions, so the inserted data and rules that were present in such a module are restored when the system restarts and the module is re-attached to the database.

This package connects to databases via the ODBC interface, so the ODBC drivers for the chosen DBMS (database management system) must be installed.

The `persistentmodules` package is experimental and has been tested only with MySQL and PostgresSQL on Linux and Windows. It is possible that it also works with Oracle, DB2, and SQL Server.

3.1 PM Interface

A module becomes persistent by executing a statement that associates the module with an ODBC data source described by a DSN (*data source name*). To start using the module persistence feature, first load the following package into some module. For instance:

```
?- [persistentmodules>>pm].
```

The following API is available. Note that `pm` is just a name we chose for the sake of an example. If you load `persistentmodules` into some other module, say `foo`, then `foo` should be used instead of `pm` in the examples below.

- `?- ?Module[attach(?DSN,?DB,?User,?Password)]@pm.`
  This action associates the data source described by an ODBC `DSN` with the module.

  Not all DBMSs support the operation of replacing the DSN's database at run time through ODBC (the USE statement). For instance, MS Access or PostgresSQL do

not. In this case, `?DSN` must be bound to `connectDSN + dbDSN` (separate DSNs for connecting to DBMS and for the actual work database), as explained in Section 3.2.

For other DBMS, such as MySQL, SQL Server, and Oracle, `?DSN` can be just an atom that specifies the work database.

The `?User` and `?Password` must be bound to the user name and the password to be used to connect to the database.

The database specified by the DSN must already exist and must be created by a previous call to the method `attachNew` described below. Otherwise, the operation is aborted. The database used in the `attach` statement must not be accessed directly—only through the persistent modules interface. The above statement will create the necessary tables in the database, if they are not already present.

Note that the same database can be associated with several different modules. The package will not mix up the facts that belong to different modules.

- `?- ?Module[attachNew(?DSN,?DB,?User,?Password)]@pm.`
  Like `attach`, but a new database is created as specified by `?DSN`. If the same database already exists, an exception of the form `FLORA_DB_EXCEPTION(?ErrorMsg)` is thrown. (In a program, include `flora_exceptions.flh` to define `FLORA_DB_EXCEPTION`; in the shell, use the symbol `'_$flora_db_error'`.) This method creates all the necessary tables, if they are not already present.

  As with the `attach` method, some DDBMS, like PostgresSQL, will need the DSN to be specified as `connectDSN + dbDSN`.

  As with the `attach` method, some DDBMS, like PostgresSQL, will need the DSN to be specified as

  Note that this command works only with database systems that understand the SQL command `CREATE DATABASE`. For instance, MS Access does not support this command and will cause an error.

- `?- ?Module[detach]@pm.`
  Detaches the module from its database. The module is no longer persistent in the sense that subsequent changes are not reflected in any database. However, the earlier data is not lost. It stays in the database and the module can be reattached to that database.

- `?- ?Module[loadDB]@pm.`
  On re-associating a module with a database (i.e., when `?Module[attach(?DSN, ?DB,?User,?Password)]@pm` is called in a new $\mathcal{F}$LORA-*2* session), database facts previously associated with the module are loaded back into it (but rules are loaded). However, since the database may be large, $\mathcal{F}$LORA-*2* does not preload it into the main memory. Instead, facts are loaded on-demand. If it is desired to have all these facts in main memory at once, the user can execute the above command. If no previous association between the module and a database is found, an exception is thrown.

- `?- set_field_type(?Type)@pm.`
  By default, $\mathcal{F}$LORA-*2* creates tables with the VARCHAR field type because this is the only type that is accepted by all major database systems. However, ideally, the CLOB (character large object) type should be used because VARCHAR fields are limited to

4000-7000 characters, which is usually inadequate for most needs. Unfortunately, the different database systems differ in how they support CLOBs, so the above call is provided to let the user specify the field types that would be acceptable to the system(s) at hand. The call should be made **before** the method `attachNew` is used. Examples:

```
?- set_field_type('TEXT DEFAULT NULL')@pm.   // MySQL, PostgreSQL
?- set_field_type('CLOB DEFAULT NULL')@pm.   // Oracle, DB2
```

Once a database is associated with the module, querying and insertion of the data into the module is done as in the case of regular (transient) modules. Therefore PM's provide a transparent and natural access to the database and every query or update may, in principle, involve a database operation. For example, a query like `?- ?D[dept -> ped]@StonyBrook.` may invoke the SQL `SELECT` operation if module `StonyBrook` is associated with a database. Similarly `insert{a[b -> c]@stonyBrook}` and `delete{a[e -> f]@stonyBrook}` will invoke SQL `INSERT` and `DELETE` commands, respectively. Thus, PM's provide a high-level abstraction over the external database.

Note that if `?Module[loadDB]@pm` has been previously executed, queries to a persistent module will *not* access the database since $\mathcal{F}$LORA-*2* will use its in-memory cache instead. However, insertion and deletion of facts in such a module will still cause database operations.

3.2 More on DSNs

ODBC drivers to DBMS's can be divided into two categories:

1. those that support the USE statement that can change database connection at run time (MySQL, Oracle, others); and
2. those that do not (PostgreSQL, MS Access).

For the first category, a DSN can be as simple as

```
[mydb]
Driver   = MySQL
Server   = localhost
User     = xsb
```

Note that the actual database is not specified, as the package sets it at run-time. The above DSN is for MySQL; other DBMS in the same category may require additional fields in a DSN.

Here is how the attachment commands look in this case:

```
?- foo[attachNew(mydb, foo, xsb, 'xyz%56G9U')]@pm.
?- foo[attach(mydb, foo, xsb, 'xyz%56G9U')]@pm.
```

For the second category of DBMS, one needs to define two DSNs: the first for connecting to the DBMS and creating the work database, and the second for working with that database. This is how they might look for PostgreSQL:

```
[postw]
# This is the DSN for connecting to PostgreSQL
Driver   = PostgreSQL_Unicode
User     = postgres
Database = template1
# For Postgres must be Servername, not Server!!!
Servername = localhost

[postw2]
# This is the DSN for actually working with the database xsb2 to which a
# PM is attached
Driver   = PostgreSQL_Unicode
User     = postgres
Database = xsb2
# For Postgres must be Servername, not Server!!!
Servername = localhost
```

Here the `Database` field is mandatory. The database in the first DSN, `template1`, is an existing DBMS, which we used for the first connection in which the package will create the work database `xsb2`. The second DSN uses the work database `xsb2`. We could not use that DSN for connection because in `attachNew` the database `xsb2` did not exist yet. (MySQL does not have that problem, as it allows one to omit the `Database` field.)

This is how the attachment commands look like in the second case:

```
?- foo[attachNew(postw+postw2, xsb2, postgres, 'xyz%56G9U')]@pm.
?- foo[attach(postw+postw2, xsb2, postgres, 'xyz%56G9U')]@pm.
```

3.3 Examples

The following example puts everything together. This example works with MySQL-like databases. For Postgres, use the attachment commands shown above.

```
// Create new modules mod, db_mod1, db_mod2.
flora2 ?-  newmodule{mod}, newmodule{db_mod1}, newmodule{db_mod2}.
flora2 ?-  [persistentmodules>>pm].

// pre-insert preliminary data into all three modules.
// Once db_mod1/db_mod2 are attached to a DB, the pre-inserted data will be
// added to the database (but not pre-inserted rules)
flora2 ?-  insert{q(a)@mod,q(b)@mod,p(a,a)@mod}.
flora2 ?-  insert{p(a,a)@db_mod1, p(a,b)@db_mod1}.
flora2 ?-  insert{q(a)@db_mod2,q(b)@db_mod2,q(c)@db_mod2}.

//  Associate modules db_mod1, db_mod2 with an EXISTING database db
//  To attach db_mod1/db_mod2 to a NEW database, use attachNew(...)
```

```
//  The data source is described by the DSN mydb.
flora2 ?-  db_mod1[attach(mydb,db,myuser,mypwd)]@pm.
flora2 ?-  db_mod2[attach(mydb,db,myuser,mypwd)]@pm.

// insert more data into db_mod2 and mod.
flora2 ?-  insert{a(p(a,b,c),d)@db_mod2}.
flora2 ?-  insert{q(a)@mod,q(b)@mod,p(a,a)@mod}.

// Both pre-inserted and the latest data will now be in the database

// shut down the engine
flora2 ?-  \halt.
```

Restart the $\mathcal{F}$LORA-*2* system.

```
// Create the same modules again
flora2 ?-  newmodule{mod}, newmodule{db_mod1}, newmodule{db_mod2}.

// try to query the data in any of these modules.
flora2 ?-  q(?X)@mod.
No.

flora2 ?-  p(?X,?Y)@db_mod1.
No.

//  Attach the earlier database to db_mod1.
flora2 ?-  [persistentmodules>>pm].
flora2 ?-  db_mod1[attach(mydb,db,user,pwd)]@pm.

// try querying again...

// Module mod is still not associated with any database and nothing was
// inserted there even transiently, we have:
flora2 ?-  q(?X)@mod.
No.

// But the following query retrieves data from the database associated
// with db_mod1.
flora2 ?-  p(?X,?Y)@db_mod1.
?X = a,
?Y = a.

?X = a,
?Y = b.

Yes.
```

```
// Since db_mod2 was not re-attached to its database,
// it still has no data, and the query fails.
flora2 ?-  q(?X)@db_mod2.

No.
```

3.4 Unsupported Features

Insertion and deletion of rules into persistent modules is supported, but insertion/deletion of latent queries into such modules might not be fully supported (this should work but has not been tested extensively).

Unlike facts inserted into modules, if a module had rules inserted into it before the module is attached to a database, those rules will not be automatically inserted into that database and thus will not persist from session to session.

Also, static rules—those appearing in loaded files (as opposed to the added files)—cannot be made persistent.

Finally, disabled rules that are saved in a database get re-enabled after they are loaded back into main memory. That is, rule disablement is not persistent.

Chapter 4

SGML and XML Import for $\mathcal{F}$LORA-*2*

by Rohan Shirwaikar and Michael Kifer

This chapter documents the $\mathcal{F}$LORA-*2* package that provides XML and XPath parsing capabilities. The main predicates support parsing SGML, XML, and HTML documents, and create $\mathcal{F}$LORA-*2* objects in the user specified module. Other predicates evaluate XPath queries on XML documents and create $\mathcal{F}$LORA-*2* objects in user specified modules. The predicates make use of the **sgml** and **xpath** packages of XSB.

4.1 Introduction

This package supports parsing SGML, XML, and HTML documents, converting them to sets of $\mathcal{F}$LORA-*2* objects stored in user-specified $\mathcal{F}$LORA-*2* modules. The SGML interface provides facilities to parse input in the form of files, URLs and strings (Prolog atoms).

For example, the following XML snippet

```
<greeting id='1'>
<first ssn=111'>
John
</first>
</greeting>
```

will be converted into the following $\mathcal{F}$LORA-*2* objects:

```
obj1[ greeting -> obj2]
obj2[ attribute(id) -> '1']
obj2[ first -> obj3]
obj3[ attribute(ssn) -> '111']
obj3[ \text -> 'John']
```

To load the `XML` package, just call any of the API calls at the $\mathcal{F}$LORA-*2* prompt.

The following calls are provided by the package. They take SGML, XML, HTML, or XHTML documents and create the corresponding $\mathcal{F}$LORA-*2* objects as specified in Section 4.3.

`?InDoc[load_sgml(?Module) -> ?Warn]@\xml`
: Import XML data as $\mathcal{F}$LORA-*2* objects.

`?InDoc[load_xml(?Module) -> ?Warn]@\xml`
: Import SGML data as $\mathcal{F}$LORA-*2* objects.

`?InDoc[load_html(?Module) -> ?Warn]@\xml`
: Import HTML data as $\mathcal{F}$LORA-*2* objects.

`?InDoc[load_xhtml(?Module) -> ?Warn]@\xml`
: Import XHTML as $\mathcal{F}$LORA-*2* objects.

The arguments to these predicates have the following meaning:

`?InDoc` is an input SGML, XML, HTML, or XHTML document. It must have one of these forms: `url('`*url*`')`, `file('`*file name*`')` or `string('`*document as a string*`')`. If `?InDoc` is just a plain Prolog atom ($\mathcal{F}$LORA-*2* symbol) then `file(?Source)` is assumed. `?Module` is the name of the $\mathcal{F}$LORA-*2* module where the objects created by the above calls should be placed; it must be bound. `?Warn` gets bound to a list of warnings, if any are generated, or to an empty list; it is an output variable.

4.2 Import Modes for XML in Ergo

XML can be imported into $\mathcal{F}$LORA-*2* in several different ways, which can be specified via the `set_mode(...)@\xml` primitive. These modes control two aspects of the import:

- white space handling, and
- navigation links that may be added to the imported data.

4.2.1 White Space Handling

The XML standard requires that white space (blanks, tabs, newlines, etc.) must be preserved by XML parsers. However, in the applications where $\mathcal{F}$LORA-*2* is used, XML typically is viewed as a format for data in which white space is immaterial. For that reason, by default, the $\mathcal{F}$LORA-*2*'s XML parser operates in the *data mode* in which every string is trimmed on both sides to remove the white space. In addition, the empty strings `''` are ignored. This implies that, for example, there will be no `\text` attribute to represent a situation like this:

```
<doc>
    <spaceonly>   </spaceonly>
</doc>
```

and the only data created to represent the above document will be

```
obj1[doc->obj2]@bar
obj2[spaceonly->obj3]@bar
```

(plus some additional navigational data about order, siblings, parents, etc.). This means that, if capturing certain white space is needed, it should be encoded explicitly in some way, e.g.,

```
<spaceonly>___</spaceonly>
```

instead of three spaces.

Alternatively, one can request to change the XML parsing mode to *raw*:

```
?- set_mode(raw)@\xml.
```

In this case, the parser will switch to the pedantic way XML parsers are supposed to interpret XML and all white space will be preserved. However, beware what you wish because even for the above tiny example the representation will end up not pretty because every little bit of white space will be there (even the one that comes from line breaks):

```
obj1[doc->obj2]@bar
obj2[\text->obj3]@bar
obj2[\text->obj6]@bar
obj2[spaceonly->obj4]@bar
obj3[\string->'
  ']@bar
obj4[\text->obj5]@bar
obj5[\string->'   ']@bar
obj6[\string->'
']@bar
```

It is more than likely an $\mathcal{F}$LORA-*2* user will not want objects like `obj3` and `obj6`.

Finally, if the *raw* mode is not what is desired, one can always switch back to the data mode:

```
?- set_mode(data)@\xml.
```

4.2.2 Requesting Navigation Links

This aspect can be changed via the calls

```
?- set_mode(nonavlinks)@\xml.  // the default
?- set_mode(navlinks)@\xml.
```

where `nonavlinks` is the default.

The `nonavlinks` method uses a slightly simpler translation from XML to $\mathcal{F}$LORA-*2* objects and no extra navigation links are provided. This mode is used when the imported XML

document has known tructure and is viewed simply as set of data to be ingested (e.g., payroll data).

In the `navlinks` mode, the representation is slightly more complex but, most importantly, that imported data includes additional information that provides parent/child/sibling links among XML objects as well as the ordering information, which allows one to reconstruct the original XML document. This mode is used when the structure of the input XML has high variability or may even be arbitrary. This arises, for instance, when one needs to transform arbitrary XML import or to extract certain information from unknown structures. The exact representation of this navigational information is described in subsequent sections.

4.3 Mapping XML to *FLORA-2* Objects

This mapping is based on an XML-to-*FLORA-2* object correspondence developed by Guizhen Yang. It specifies how an XML parser can construct the corresponding F-logic objects after parsing an input XML document. The basic ideas are as follows:

- XML elements, attribute values, and text strings are modeled as objects in F-logic.
- XML elements are reachable from parent objects via F-logic frame attributes of the same name as the XML element name.
- XML element attributes are also modeled as F-logic frame attributes but their name is `attribute(`*XML attribute name*`)`.

This mapping does not address comments or processing instructions—they are simply ignored. However, this mapping does address the issue of mixed text/element content in which plain text and subelements are interspersed. This mapping also assumes that XML entities are resolved by the XML parser.

4.3.1 Invention of Object Ids for XML Elements

According to the XML specification 1.0, an XML element can be identified by an oid that is unique across the document. The import mechanism invents such an oid automatically. Sitting on top of the XML root element, there is an additional root object which just functions as the access point to the entire object hierarchy corresponding to the XML document. The oids of leaf nodes, which have no outgoing arcs and carry plain text only, are just the string values themselves.

For example, the following XML document

```
<?xml version="1.0"?>
<person ssn="111-22-3333">
  <name first="John"
        last="Smith"/>
</person>
```

is represented via the following F-logic objects:

```
obj1[person -> obj2].
obj2[attribute(ssn) -> '111-22-3333', name -> obj3].
obj3[attribute(first) -> John, attribute(last) -> Smith].
```

Here `obj1` is the root object, `obj2` is the object corresponding to the `person` element, and `obj3` is the object that represents the `name` element. The strings `'111-22-3333'`, `John`, and `Smith` are oids that stand for themselves.

4.3.2 Text and Mixed Element Content

The content of an XML element may consist of plain text, or subelements interspersed with plain text as in

```
<greeting>Hi! My name is <first>John</first><last>Smith</last>.</greeting>
```

How text is actually handled in the translation to F-logic depends on the mode of import: `nonavlinks` or `navlinks`. The former is simpler because it discards all the information about the order of the text nodes with respect to subelements and other text nodes.

- **In the `nonavlinks` mode**:
  Each text segment is modeled as a value of the attribute `\text` of the parent element-object of that text segment.[1] Thus, for the above XML fragment, the translation would be

  ```
  obj1[greeting -> obj2].
  obj2[\text -> {'Hi! My name is ', '.'},
       first -> obj3,
       last  -> obj4
  ].
  obj3[\text -> John].
  obj4[\text -> Smith].
  ```

- **In the `navlinks` mode**:
  Here the order of the text and subelement nodes must be preserved and so each text node is modeled as if it were a value of a special attribute `\string` in an empty XML element named \text, e.g.,

  ```
  <\text \string="John"/>
  ```

  As a consequence, a separate F-logic object is created to represent each text segment. (Compare this to the translation in the `nonavlinks` mode, which does not create separate objects for text nodes.) Thus, for the aforesaid `greetings` element the translation will be

[1] Of course, XML does not allow such names for tags and attributes, and this is the whole point: adding such an invented name to the F-logic translation will not clash with other tag names that might be used in the XML documents.

```
obj1[greeting -> obj2].
obj2[\text -> {obj3, obj8},
     first -> obj4,
     last  -> obj6
].
obj3[\string -> 'Hi! My name is '].
obj4[\text -> obj5].
obj5[\string -> John].
obj6[\text -> obj7].
obj7[\string -> Smith].
obj8[\string -> '.'].
```

How exactly the aforesaid order is preserved in the `navlinks` mode is explained later.

4.3.3 Translation of XML Attributes

An XML attribute, *attr*, in an element is translated as an attribute by the name `attribute`(*attr*) attached to the object that corresponds to that element.

XML element attributes of type IDREFS are multivalued, in the sense that their value is a string consisting of one or more oids separated by whitespaces. Therefore, the value of such an attribute is a set. The value of an XML IDREFS attribute is represented as a list.

For example, the following XML segment:

```
<paper id="yk00" references="klw95 ckw91">
  <title>paper title</title>
</paper>
```

will generate the following F-logic atoms, assuming that the `reference` attribute is of type IDREFS:

```
obj1[paper -> obj2]
obj2[title -> obj4]
obj2[attribute(id) -> yk00]
obj2[attribute(references) -> 'klw95 ckw91'
obj4[\text -> obj5]      // here we assume that the navlinks mode was used
obj5[\string -> 'paper title']
```

However: if the document has an associated DTD *and* the attribute `references` were specified there as `IDREFS` as in

```
<!ATTLIST paper references IDREFS #IMPLIED>
```

then that attribute is translated as

```
obj2[attribute(references)->[klw95,ckw91]]
```

i.e., the value becomes a list.

With this, we are done describing the `nonavlinks` mode. The remaining subsections in the current section apply to the `navlinks` mode only.

4.3.4 Ordering

This section applies to the `navlinks` mode only.

XML is order-sensitive and the order in which elements and text appear is significant, in general. The order of the attributes within the same element tag is *not* significant, however.

While the `nonavlinks` mode is sufficient for most data-intensive uses of XML in *FLORA-2*, more complex tasks may require the knowledge of how items are ordered within XML documents. Specifying a total order among the elements and text in an XML document suffices for that purpose, if this order agrees with the local order within each element's content.

Consider the following XML document

```
<?xml version="1.0"?>
<person ssn="111-22-3333">
  <name>
    <first>John</first>
    <last>Smith</last>
  </name>
  <email>jsmith@abc.com</email>
</person>
```

It can be represented by the tree in Figure 4.1 in which the parenthesized integers show the total order assigned to the F-logic objects.

The ordering information that exists in XML documents is captured in F-logic via a special attribute called `\order`, which tells position within the total ordering for each element and text node. It is for that purpose that text segments are modeled in the `navlinks` mode as element-style objects (each segment having its own oid) and not simply as attributes, as is the case with the simpler `nonavlinks` mode.

4.3.5 Additional Attributes and Methods in the `navlinks` Mode

Since the `navlinks` mode is intended for applications that need to navigate from children to parents, to siblings, and more, the importer adds the following additional attributes and methods to the F-logic objects into which XML elements and text are mapped.

1. `\in_arc`
   For each node, `\in_arc` returns the unordered set of labels of the arcs pointing to this node, i.e., this node's in-arcs. Roughly, `\in_arc` is defined as follows:

   ```
   ?O[\in_arc -> ?InArc] :- ?[?InArc -> ?O].
   ```

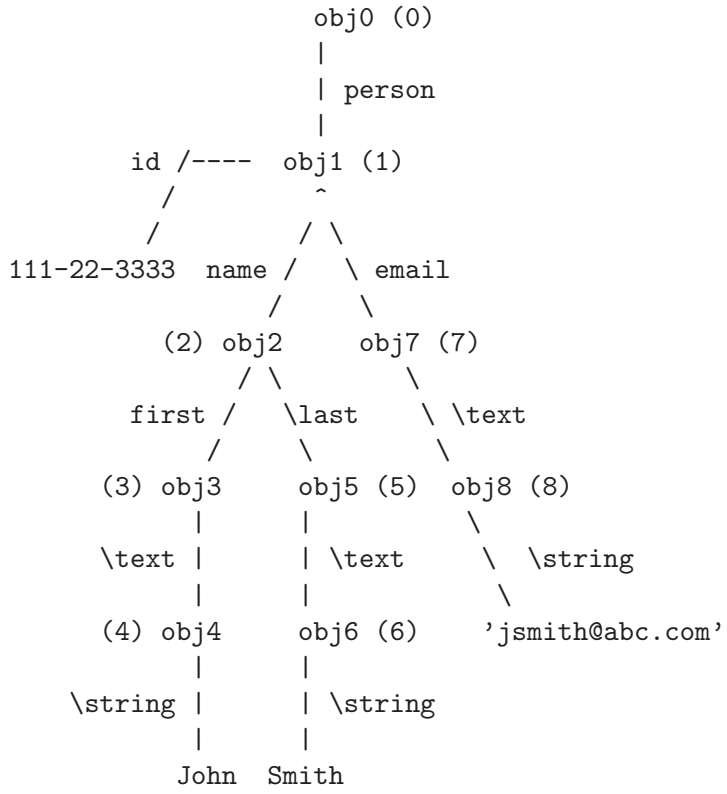

Figure 4.1: Total ordering of the F-logic objects arising from XML ordering

Note that for a node representing a text segment, the value of its \in_arc attribute is \text.

2. \parent
   For each node, \parent returns the oid of the parent node.

3. \leftsibling
   For each node, \leftsibling returns the oid of the node appearing immediately before the current node. This attribute is not defined for the nodes without a left sibling.

4. \rightsibling
   For each node, \rightsibling returns the oid of the node appearing immediately after the current node. This attribute is not defined for the nodes without a right sibling.

5. \childcount
   For each element node, \childcount returns the number of the immediate children of that element, which includes subelements and text segments.

6. \childlist
   For each element node, \childlist returns a list of the oids of the immediate children (subelements and text segments) of that element.

7. \child(N)

For each node, \child(N) returns the N-th child, where 0 ≤ N < \childcount. Note: the first child is the 0-th child.

8. \in_child_arc(N)
   For each node, \in_child_arc(N) returns the in-arcs of the N-th child, where 0 ≤ N < \childcount. This attribute is defined as follows:

```
?O[\in_child_arc(?N)->?InArc] :- ?O[\child(?N)->?[\in_arc->?InArc]].
```

4.4 Inspection Predicates

This section applies both to the nonavlinks mode and the navlinks mode.

It is sometimes hard to see which objects have actually been created to represent an XML document or an element. This is especially true in case of navlinks mode, which includes a host of special navigational attributes. The purpose of inspection predicates is to provide a simple way to view the objects, and they also filter the navigational attributes out. Consider the document foo.xml below:

```
<mydoc id='1'><first ssn='111'>John</first></mydoc>
```

Even for such a simple document, the query

```
?- 'foo.xml'[load_xml(bar) -> ?W]@\xml.    // load foo.xml into module bar
?- ?_X[?_Y->?_Z]@bar, ?Z = ${?_X[?_Y->?_Z]}.  // get all facts
```

that asks for all the facts—stored and derived—will yield 56 results in the navlinks mode, which is overwhelming to inspect visually. However, the core facts that describe these objects are only 8, and they can be obtained by asking the query

```
?- bar[show->?P]@\xml.
```

One furthermore might want to see the representation of individual elements (e.g., element named first):

```
?- bar[show(first)->?P]@\xml.
```

and this is much more manageable:

```
?P = ${obj4[\text->obj5]@bar}
?P = ${obj4[attribute(ssn)->'111']@bar}
```

or of elements that have particular attributes (ssh in this example):

```
?- bar[show(attribute(ssn))->?P]@\xml.
```

which yields the same result as above (because the element first has the attribute ssn).

4.5 XPath Support

The XPath support is based on the XSB `xpath` package, which must be configured as explained in the XSB manual. This package, in turn, relies on the XML parser called `libxml2`. It comes with most Linux distributions and is also available for Windows, MacOS, and other Unix-based systems from `http://xmlsoft.org`. Note that both the library itself and the `.h` files of that library must be installed.

Note: XPath support does not currently work under Windows 64 bit due to the fact that we could not produce a working `libxml2.lib` file (`xmlsoft.org` provides `linxml2.dll` for Windows 64, but not `libxml2.lib`).

The following predicates are provided. They select parts of the input document using the provided XPath expression and create $\mathcal{F}$LORA-*2* objects as specified in Section 4.3. These predicates handle XML, SGML, HTML, and XHTML, respectively.

`?InDoc[xpath_xml(?XPathExp,?NS,?Mod)->?Warn]`	apply XPath expression to an XML document and import the result
`?InDoc[xpath_xhtml(?XPathExp,?NS,?Mod)->?Warn]`	apply XPath expression to XHTML and import the result

The arguments have the following meaning:

`InDoc` specifies the input document; this parameter has the same format as in Section 4.1. `?XPath` is an XPath expression specified as a Prolog atom. `?Module` is the module where the resulting $\mathcal{F}$LORA-*2* objects should be placed. `?Module` must be bound. `?Warn` gets bound to a list of warnings, if any are generated during the processing—or to an empty list, if none.

`?NamespacePrefList` is a string that has the form of a space separated list of items of the form *`prefix = namespaceURL`*. This allows one to use namespace prefixes in the `?XPath` parameter. For example if the XPath expression is `'/x:html/x:head/x:meta'` where `x` stands for `'http://www.w3.org/1999/xhtml'`, then this prefix would have to be defined in `?NamespacePrefList`:

```
url('http://w3.org')[xpath_xhtml('/x:html/x:head/x:meta',
                                 'x=http://www.w3.org/1999/xhtml',
                                 foomodule)
                     -> ?Warnings]@\xml.
```

4.6 Low-level Predicates

This section describes low-level predicates in the XML package. These predicates parse the input documents into Prolog terms that then must be further traversed recursively in order to get the desired information.

- `parse_structure(?InDoc,?InType,?Warnings,?ParsedDoc)@\xml` — take the document `?InDoc` or type `?InType` (`xml`, `xhtml`, `html`, `sgml`) and parse it as a Prolog term (will not be imported into any module as an object).

- `apply_xpath(?InDoc,?InType,?XPathExp,?NamespacePrefList,?Warnings,?ParsedDoc)@\xml` — like the above, but first applies the XPath expression `?XPathExp` to `?InDoc`. The `?InType` parameter must be bound to `xml` or `xhtml`.

The output, `?ParsedDoc`, is a Prolog term that represents the parse of the input XML document in case of `parse_structure` and the result of application of `?XPathExp` to the input document in case of `apply_xpath`. The format of that parse is described in the *XSB Manual, Volume 2: Interfaces and Packages*, in the chapter on *SGML/XML/HTML Parsers and XPath*.

Bibliography

[1] Miguel Calejo. Interprolog: Towards a declarative embedding of logic programming in java. In Jóse Júlio Alferes and João Leite, editors, *Logics in Artificial Intelligence: 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004. Proceedings*, pages 714–717. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.