Connecting to a Relational Database Application from Interleaf 5 Using Lisp

Charles Dale
Member of the Technical Staff, Interleaf, Inc.

The Interleaf 5 (I5) desktop publishing software provides a powerful customization environment based on Lisp. A small development team, including the author, used this environment to build a tightly integrated I5 user interface (UI) for Interleaf's Relational Document Manager (RDM). This paper describes the part of this program that implements the connection between I5 Lisp and the RDM Application Program Interface (API).

RDM is a document management system that supports large documentation projects by providing the following three major pieces of functionality:

1. Librarian services such as document storage and retrieval, and the tracking of multiple drafts and revisions
2. Workflow management, including document object routing and a message system that provides notification of incoming tasks
3. Dynamic access control based on the document's step in its workflow

Users build a model within RDM of their documentation project, including the structure of the various books being produced and the organizational structure of the documentation group (writers, editors, reviewers, etc.). RDM objects are defined to represent individual documents or document hierarchies (books). Each RDM object is assigned an object type that specifies its development cycle or routing. The routing is a sequence of steps that an RDM object must pass through during its development cycle. Each step of the routing is associated with one or more RDM users. Users and groups are defined in RDM's organizational structure model. RDM objects are stored in (usually protected) directories called vaults.

All of these entities — objects, object types, users, groups, and vaults — are stored in tables in a relational database. The main RDM application is a C program that includes embedded SQL statements to access and modify the database tables. This program has no UI, but provides an API that can be connected to UI front ends. We

have implemented three front ends, one using a portable GUI toolkit called JAM that runs under X and Windows, one using HyperCard, and one using the I5 desktop UI.

Figure 1 illustrates this architecture.

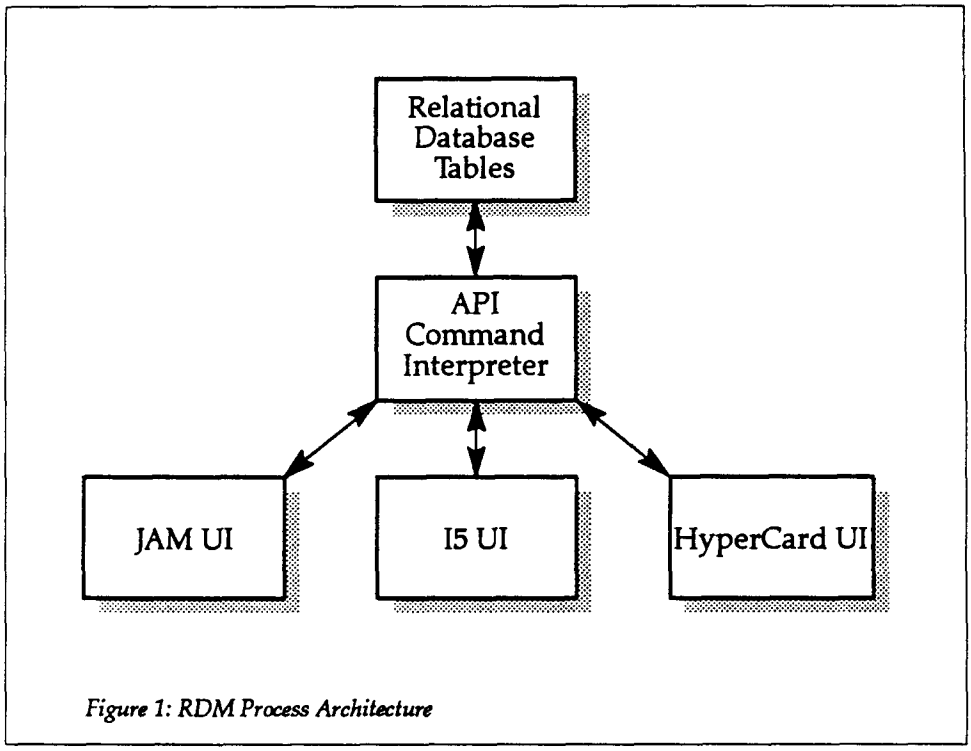

Figure 1: RDM Process Architecture

The design of the RDM API was strongly influenced by the Lisp `read-eval-print` loop. This influence is evident in the fact that the RDM Command Interpreter includes limited Lisp READ and PRINT modules capable of handling atoms (without package prefixes), strings (with slashification), numbers, and nested list structure. As will be seen, this is sufficient to communicate the entire range of RDM requests and responses. With READ and PRINT as the common denominator, communications between the Command Interpreter and I5 are established using character streams. The first time a user invokes RDM, I5 automatically starts the Command Interpreter as a separate process and connects its standard input and standard output to I5 Lisp streams. I5 then makes RDM calls using `print` on the Command Interpreter's input stream and receives the returned data using `read` on its output stream.

Input to the Command Interpreter is of the form (*request data-list*). Some requests need no input and so do not need a *data-list*. Each *request* corresponds to a C function that takes a data list as an argument, interprets the various elements of the data list, and calls the appropriate database interface function with the arguments

parsed from the data list. It is also responsible for signaling errors caused by the wrong number of arguments or incorrect argument types.

Values are returned from the Command Interpreter in the form (*error data-list*). If the request caused an error, *error* is a string describing the problem; otherwise it is nil. If the request was a query, the selected data is returned in *data-list*. For updates, *data-list* is nil. The format of *data-list* depends on the specific request, but it is generally a list of lists, one per row from the SQL SELECT statement. Each query request defines the columns or fields returned in these sublists. Each field is a string or, in some cases, a list. Example 1 shows a typical API call and the returned values.

Example 1: API Input and Output

```
Input:    (get-user-detail ("CTD"))
Output:   (nil
           (("CTD" "USR" "" "CHARLES DALE" "ADMIN" "System Administrator" "SYSTEM"
             "SYSTEM TOP LEVEL" "" "/u/monteverdi/ctd/desktop" "ctd" "")))
```

In this example, the request `get-user-detail` takes one argument, `"CTD"`, a user-id. The rows that it returns contain 12 columns. In this case, only one row was returned.

To make it easier to deal with the lists returned from the Command Interpreter, we use a macro that constructs a `defstruct` to define list structures and provide accessor functions. The structure returned in Example 1 could be defined as follows:

Example 2: The `user` API Data Structure

```
(define-api-structure user
  name type password full-name default-function function-description parent-group
  group-description last-message work-area mail-address group-vault)
```

While the above mechanism works, it would be very cumbersome to write a large program using it. There are two major problems. First, every time you make an RDM request, you have to check the *error* value and handle it if it is non-nil. For functions that make several RDM requests, a considerable amount of code would have to be devoted to handling API errors. This not only detracts from the readability of the code, but it is very repetitive and error-prone. The other problem is that the argument structure of API requests is very rigid, in that it supports only required arguments. For calls with many arguments, such as object search, this can be difficult to program and maintain.

To make access to the API more Lisp-like, we defined a Lisp API layer that mediates between the Command Interpreter and the rest of the I5 UI. For every API request, we define a Lisp function that provides the interface to it. The Lisp API function can use optional and keyword arguments and provide defaults. It also takes care of cases in which errors are returned by signaling an RDM error condition.

Figure 2 shows how the Lisp API fits into the overall RDM architecture.

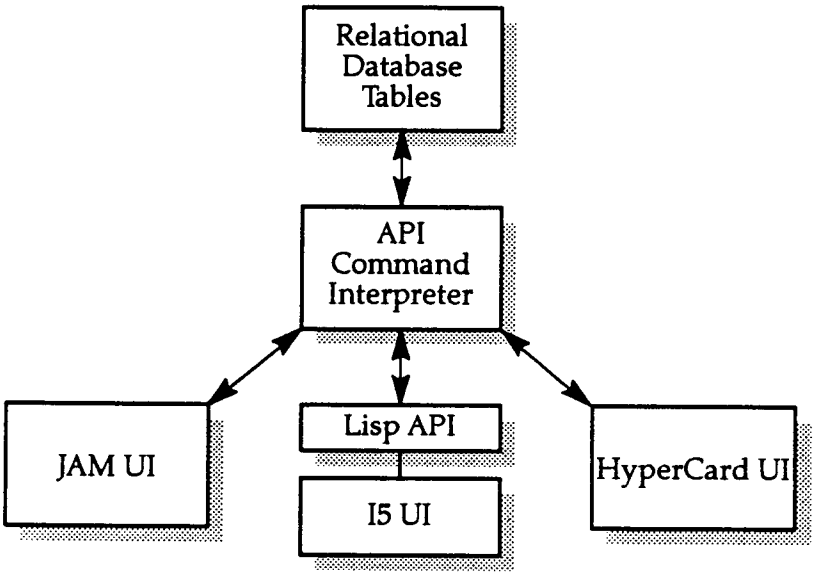

Figure 2: RDM Process Architecture with Lisp API

A macro called `handling-api-errors` is provided to make programming with Lisp API functions more transparent. The Lisp API can be called multiple times within a single `handling-api-errors` form. By default, each API call runs as a separate transaction, but calls can be grouped into a single transaction by using the `open-transaction`, `commit`, and `rollback` API calls. `handling-api-errors` catches the `rdm-api-error` condition and performs error recovery, optionally calling `rollback`. It also starts the Command Interpreter process if it is not already running. Inside `handling-api-errors`, programs use the Lisp API functions like any other Lisp functions. Example 3 shows the interface to the `get-user-detail` request in the Lisp API. Compared with Example 1, this is simpler to call, and the return values are easier to use.

Example 3: Lisp API Input and Output

```
Input:    (get-user-detail "CTD")
Output:   (("CTD" "USR" "" "CHARLES DALE" "ADMIN" "System Administrator" "SYSTEM"
            "SYSTEM TOP LEVEL" "" "/u/monteverdi/ctd/desktop" "ctd" ""))
```

In Example 3, if `get-user-detail` encounters an error, it signals an error of type `rdm-api-error` rather than return any output. Example 4 shows a simple program written using the Lisp API.

Example 4: A Simple Program

```
(defun list-all-users ()
  (handling-api-errors ()
    (terpri)
    (dolist (user (get-user-detail))
      (format t "~12A ~A~%"
              (user-name user)
              (user-full-name user)))))
```

The RDM API defines over 100 requests, but it is designed to be tailored to specific applications. To provide maximum flexibility, we have a request called `execute-sql-statement` that takes an SQL statement (to protect database integrity, only SELECTs are normally allowed) and returns the resulting rows in sublists as described above. With this request, customizers can define their own new queries, structures, and API calls. This type of interface could be extended to provide access to any database that provides an SQL interface.

Example 5 shows a program that uses the `execute-sql-statement` request. It first defines a new API structure that mirrors the list of fields to be returned. It then defines a Lisp API function that uses the `call-api` function. The contract of the `call-api` function is to format the data list, call the Command Interpreter, check for and signal errors, and return the data list from the API. In practice, the `define-api-structure` and `defun` of the Lisp API function are generated by a single macro called `define-api-call`, which also generates a documentation string and optionally exports the function symbol. The last step is to define a UI function. This looks just like the function in Example 4.

Example 5: Using the `execute-sql-statement` API Call

```
(define-api-structure my-object
  name author)

(defun get-my-objects-by-author (author)
  (call-api 'execute-sql-statement
     (format nil "SELECT object_name, object_author FROM object ~
                  WHERE object_author LIKE '~A' ~
                  ORDER BY object_name"
             author)))

(defun list-my-objects (author)
  (handling-api-errors ()
    (terpri)
    (dolist (my-object (get-my-objects-by-author author))
      (print (my-object-name my-object)))))
```

After spending a great deal of time and effort developing and debugging this code, the most persistent problem has been getting the `read` and `print` functions in the Command Interpreter to obey enough of the contract of their Lisp counterparts to keep both sides from becoming hopelessly confused. The most common error occurs because Lisp and SQL have different quoting and other special characters, and getting the escaping right in both syntaxes is tricky. In retrospect, the choice of `read` and `print` for this interface may not have been the best. It certainly made the Lisp side easy to implement, but it has added a lot of complexity to the Command Interpreter and other non-Lisp UIs.

Another problem with the current implementation is that the Lisp API is defined separately from the C API. Since different people maintain each end of this interface, it is relatively easy for them to get out of sync. One way to fix this would be to enhance the `define-api-call` macro to write the C API interface functions as well.

The Lisp environment in Interleaf 5 has allowed us to build a very powerful and flexible user interface for RDM. Since most installations require at least some amount of customization, making that it as easy as possible was a major design goal. We also expect to be able to adapt the tools we built for the RDM interface to other, more generic database applications.