SchemeUnit and SchemeQL: Two Little Languages

Noel Welsh
LShift
Burbage House
83-85 Curtain Road
London, EC2A 3BS, UK
noel@lshift.net

Francisco Solsona
Universidad Nacional
Autónoma de México
Mexico City, Mexico 04510
solsona@acm.org

Ian Glover
Cambridge Positioning
Systems
62-64 Hills Road
Cambridge, UK
ian@manicai.net

ABSTRACT

We present two little languages implemented in Scheme: SchemeUnit, a language for writing unit tests, and SchemeQL, a language for manipulating relational databases. We discuss their design and implementation and show how the features of functional languages in general, and Scheme in particular, contribute to the ease of use and implementation of our languages.

Keywords

Scheme, Little Language, SQL, Unit testing, SchemeQL

1. INTRODUCTION

The domain specific language, or little language, is a powerful technique for increasing programmer productivity. Much work in domain specific languages has been done in functional languages (e.g. [28, 13, 8]). Our work is no different in this regard. Our contribution is to focus on the interface of our languages and show how we can use the features of functional languages in general, and Scheme in particular, to improve the user experience. We describe little languages for unit testing and relational database manipulation. The two languages have been used by the authors and others in real applications, and the code is available from

http://schematics.sourceforge.net/

2. THE SCHEMEUNIT FRAMEWORK

Unit testing concerns testing individual elements of a program in isolation. SchemeUnit is a framework for defining, organizing, and executing unit tests written in the PLT dialect of Scheme[11]. We drawn inspiration from two strands of work: existing practice in interactive environments and the development of unit testing frameworks following the growth of Extreme Programming.

In an interactive environment it is natural to write in a "code a little, test a little" cycle: evaluating definitions and then immediately testing them in the read-eval-print loop (REPL). We take the simplicity and immediacy of this cycle as our model. By codifying these practices we preserve the test cases beyond the running time of the interpreter allowing the tests to be run again when code changes.

Third Workshop on Scheme and Functional Programming. October 3, 2002, Pittsburgh, Pennsylvania, USA.

Unit testing is one of the core practices of the Extreme Programming[3] software development methodology. Unit testing is not new to Extreme Programming but Extreme Programming's emphasis on unit testing has spurred the development of software frameworks for unit tests. The original unit testing framework (SUnit) is written in SmallTalk[2]. Since then unit testing frameworks have been written for many languages[18]. We draw inspiration from these frameworks and find it enlightening to compare the expressivity of these frameworks with SchemeUnit. In particular we will compare SchemeUnit to JUnit[4], an extremely popular unit testing framework for the Java language (it has been downloaded over 340,000 times at the time of writing).

We start our discussion by clarifying the goals of SchemeUnit. We then describe the framework's design and show how our goals have influenced the design. We follow with a comparison of SchemeUnit and JUnit that illustrates how the expressivity of Scheme leads to a cleaner implementation and better user experience. We finish with a discussion of related and future work.

2.1 Goals

We have three goals for SchemeUnit. Firstly we want to remain as close as possible to the "code a little, test a little" cycle we described above. Secondly we want to support the main testing patterns we encounter in practice. Finally we want to support user extensions to the testing framework.

Throughout this paper we shall use an example of simple interactive testing to illustrate our design. Suppose the user is testing the invariance of write and read. The code they may execute is given below:

```
(define data (list 1 2 3 4))

(with-output-to-file "test.dat"
  (lambda () (write data)))

(with-input-from-file "test.dat"
  (lambda () (equal? data (read))))

(delete-file "test.dat")
```

The programmer checks the test by inspecting the result of the (equal? data (read)) expression. If the result is *#t* the test has succeeded.

We shall show how this example is coded in our framework and take the simplicity of the above example as our goal.

2.2 Core Design

The *test* is the core type in our framework. A test is either a *test case*, which is a single action to test, or a *test suite*, which is a collection of tests.

$$
\begin{array}{rcl}
\text{test} & \longrightarrow & \text{test-case} \mid \text{test-suite} \\
\text{test-case} & \longrightarrow & \text{name} \times \text{action} \\
\text{test-suite} & \longrightarrow & \text{name} \times \text{tests} \\
\text{tests} & \longrightarrow & \text{listof test}
\end{array}
$$

The hierarchical arrangements of tests into suites helps the programmer organize and maintain their tests.

We represent a test action as a closure. Three ways spring to mind to signal test success or failure:

1. Indicate success by returning a non-*#f* value and failure by returning *#f*.
2. Return a datatype indicating success or failure and additional information
3. Throw an exception on failure and return normally for success

The first method has the advantage of simplicity but the disadvantage that we lose information about the cause of failure, so we discard it immediately. The other two methods are equivalent in terms of the information they can return (we can encode arbitrary information in the return value or the exception). We have several reasons for choosing the third option over the second. Firstly we wish to catch exceptions anyway to prevent an unexpected error (i.e. ones that we are not testing for) from halting the testing framework. Secondly when using the second method and testing a sequence of expressions it is necessary to use continuation passing style to propagate a test failure that occurs in an intermediate expression. In this case we are simulating exceptions! Therefore for simplicity of implementation and use we choose to throw an exception to signal an error. We also divide the types of exception we catch into those we catch as the result of a tested failure (which we call *failures*) and those we catch due to untested failures (which we call *errors*).

We provide a *run-test-case* function that takes a *test-case* and returns a *test-result*:

$$\text{(run-test-case test-case)} \quad \Rightarrow \quad \text{test-result}$$

$$
\begin{array}{rcl}
\text{test-result} & \longrightarrow & \text{test-failure test-case} \times \text{failure-exn} \\
& & \mid \text{test-error test-case} \times \text{error-exn} \\
& & \mid \text{test-success test-case} \times \text{result}
\end{array}
$$

Finally, the two functions *fold-test* and *fold-test-results* make it easy to walk over tests.

$$
\begin{array}{rcl}
\text{(fold-test test-collector seed test)} & \Rightarrow & \text{seed} \\
\text{(fold-test-results result-collector seed test)} & \Rightarrow & \text{seed}
\end{array}
$$

$$
\begin{array}{rcl}
\text{seed} & \longrightarrow & \alpha \\
\text{test-collector} & \longrightarrow & (\text{test } \alpha) \longrightarrow \alpha \\
\text{result-collector} & \longrightarrow & (\text{test-result } \alpha) \longrightarrow \alpha
\end{array}
$$

2.3 Testing Patterns

Our example in the core framework is:

```
(make-test-case
 (assert binary-predicate actual expected)
 "write/read invariance"
 (lambda ()
   (let ((data (list 1 2 3 4)))
     (dynamic-wind
      (lambda ()
        (with-output-to-file "test.dat"
          (lambda () (write data))))
      (lambda ()
        (with-input-from-file "test.dat"
          (lambda ()
            (let ((actual (read)))
              (if (not (equal? actual data))
                  (raise
                   (make-exn:test:assertion
                    (string-append
                     "write/read invariance failed with "
                     (format "actual ~a" actual)
                     " and "
                     (format "expected ~a" data))))
                  #t)))))
      (lambda () (delete-file "test.dat"))))))
```

Clearly we have lost the simplicity of the original REPL! By adding common testing patterns to SchemeUnit we show how we can regain this simplicity.

2.3.1 Assertions

Checking actual output against expected output is the most common test pattern. We borrow the idea of assertion functions from JUnit. An assert function tests a condition, raising a failure exception if the condition is false. The failure exception contains the location of the failed assertion, the actual and expected parameters, and an optional user specified message string.

The core functionality can be provided by a single function:

(assert binary-predicate actual expected [message])

We know from experience that it pays to provide assertions for the most common cases, so SchemeUnit provides a library of assertions:

- (assert binary-predicate actual expected [message])
- (assert-equal? actual expected [message])
- (assert-eqv? actual expected [message])
- (assert-eq? actual expected [message])
- (assert-true actual [message])

- (assert-false actual [message])
- (assert-pred unary-predicate actual [message])
- (assert-exn exn-predicate thunk [message])
- (fail [message])

Assertions are defined using the define-assertion macro:

```
(define-assertion (name param ...) expr ...)
```

The define-assertion macro expands into the definition of a macro and a function[1] that takes the given parameters and an optional message string. If the result of the expressions is *#f* the assertion raises a failure exception containing the all the information given above.

The define-assertion macro is exported so users can define their own domain-specific assertions on par with those already provided. We hope over time to accumulate libraries of specialized assertions.

2.3.2 State Management

Note that our example test uses state and hence requires initialization and cleanup code. This is fairly common and we would like to make is easier for the user to specify these actions. Borrowing again from JUnit we call this code *setup* and *teardown* actions and we augment test-case to optionally include them. So

test-case $\longrightarrow$ name × action [× setup] [× teardown]

2.3.3 Interface Enhancements

We use macros to add the repetitive lambda statements around the action, setup, and teardown expressions. We also wrap the call to action with calls to setup and teardown in the macro rather than requiring the test framework to preform this action.

Our example is now:

```
(let ((data (list 1 2 3 4)))
  (make-test-case "write/read invariance"
    (with-input-from-file "test.dat"
      (lambda ()
        (assert-equal? (read) data)))
    (with-output-to-file "test.dat"
         (lambda () (write data)))
    (delete-file "test.dat")))
```

This code is almost identical to the original example typed at the REPL. We have achieved our ease-of-use goal, and we have done so by supporting testing patterns and allowing user extensions to the testing framework.

[1] Only macros can get location information in PLT Scheme. We define the function variant as we have occasionally found uses for higher order assertions. The function variant has a * appended to its name.

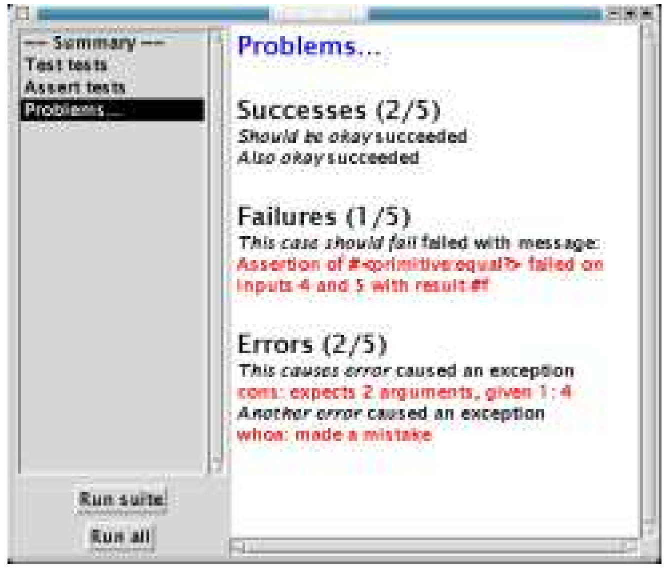

Figure 1: The SchemeUnit graphical interface

2.4 Interfaces

We provide textual and graphical interfaces to SchemeUnit. An example run shows the user interface in action. The following test suite

```
(test/text-ui
  (make-test-suite "Example suite"
    (make-test-case "Will succeed"
      (assert-equal? (+ 1 2) 3))
    (make-test-case "Will fail"
      (assert-equal? (+ 1 1) 3))
    (make-test-case "Will cause error"
      (assert-equal? (/ 1 0) 0))))
```

gives the output:

```
Error:
 Will cause error
 an error of type exn:application:divide-by-zero
 occurred with message: "/: division by zero"
Failure:
 Will fail
 assert-equal? failed at:  top-level 8:7
 Inputs: <2> <3>

1 success(es)  1 error(s)  1 failure(s)
```

The graphical interface is still in development. When complete it will provide source level highlighting and allow navigation to error location using DrScheme. An example of the current graphical interface is shown in Figure 1.

2.5 SchemeUnit versus JUnit

It is instructive to compare SchemeUnit with the popular JUnit test framework, as doing so serves to illustrate the expressive advantage of SchemeUnit. Our discussion centers on a basic example from [25] based on a telephone class. The Java code is:

```
public class TelephoneNumberTests extends TestCase {
  public static void main(String[] args) {
    junit.textui.TestRunner.run(suite());
  }
  public static TestSuite suite() {
    return new TestSuite(TelephoneNumberTests.class);
  }
  public TelephoneNumberTests(String testname) {
    super(testname);
  }
  public void testSimpleStringFormatting()
      throws Exception {
    // Build a complete phone number
    TelephoneNumber number ->
      new TelephoneNumber("612", "630",
                          "1063", "1623");
    assertEquals("Bad string",
                 "(612) 630-1063 x1623",
                 number.formatNumber());
  }
  public void testNullAreaCode()
      throws Exception {
    // Build a phone number without area code
    TelephoneNumber number ->
        new TelephoneNumber(null, "630",
                            "1063", "1623");
    assertEquals("Bad string",
                 "630-1063 x1623",
                 number.formatNumber());
}}
```

A translation of this to the SchemeUnit syntax is

```
(require (lib "test.ss" "schemeunit")
         (lib "text-ui.ss" "schemeunit"))

(test/text-ui
  (make-test-suite "Telephone number tests"

    (make-test-case "Simple format"
      (assert-equal? "(612) 630-1063 x1623"
        (format-number
          (make-number 612 630 1063 1623))
        "Bad String"))

    (make-test-case "No area code"
      (assert-equal? "630-1063 x1623"
        (format-number
          (make-number (void) 630 1063 1623))
        "Bad string"))))
```

There are several points to note about this example. One is the amount of typing required for this short example. The Java code is far more verbose, most notably in the setup code. This is largely a result of the type declarations and noise keywords (like return and new) required by Java. To our eyes the Scheme code is much more elegant though we recognize this is a subjective judgment.

JUnit relies extensively on reflection. Test cases are defined by prefixing the method name with `test`. This is an elegant solution to the problem that Java has no first class representation of functions but can lead to problems: JUnit uses a custom class loader that can interact unpredictably with other Java code that makes extensive use of reflection (e.g. Java remote method calls). This makes testing difficult in these environments. There is no such problem in Scheme.

In JUnit setup and teardown methods are similarly identified by name and discovered by reflection. Again first class functions reduce the complexity of the SchemeUnit framework.

In general structuring the test suites by value rather than by name makes for a simpler and more flexible system. There are fewer new conventions for the user to remember and tests can be manipulated on the fly.

2.6 Related and Future Work

SUnit has spawned a large and increasing number of testing frameworks of which SchemeUnit is one. We shall briefly consider those that are particularly relevant to SchemeUnit.

HUnit[14], an implementation for the Haskell Language, is a recent addition to the family. There are broad similarities between HUnit and SchemeUnit. Both signal failure with exceptions and both provide a number of convenience assertion functions. HUnit recognizes the importance of interface and defines infix operators that make test specification easier. The combination of lazy evaluation and infix operators achieves a similar effect to our macros. We briefly illustrate HUnit below, along with the equivalent code in SchemeUnit:

```
test1 -> 3 ~->? (1 + 2)
tests -> TestList [TestLabel "Addition" test1]
```

```
(define tests
  (make-test-suite
    "All tests"
    (make-test-case "Addition" (assert -> 3 (+ 1 2)))))
```

LIFT[20], CLUnit[1] (Common Lisp) and CurlUnit[5] (Curl) are Lisp dialect implementations of the SUnit framework. All are broadly similar to SchemeUnit. Both LIFT and CLUnit have some stateful features to ease interactive development of tests. Defining a test in LIFT (with deftest) implicitly creates a test suite to which later tests (created with addtest) are automatically added. In CLUnit tests are categorized by name and stored in a global collection. Tests override existing tests with the same name and are removed with the remove-test function. CurlUnit is a direct translation of JUnit to Curl so most of our earlier comments about JUnit apply to CurlUnit.

The FORT[9] framework, implemented in O'Caml, takes a different approach to the SUnit family. Test results take one of seven values including unexpected success, expected failure, untested, and unresolved in addition to the more usual pass and fail. Test results and returned by the normal function return mechanism so we envisage some difficulty in constructing a single test case containing multiple test expressions. The multitude of test results is an interesting idea but we have yet to encounter a situation where they are necessary. Lacking a clear need we favor simplicity and stick with our three result types.

As the Extreme Programming community evolved from the design pattern community it is no surprise that testing patterns[23][10] have been developed. We intend to analyze these patterns and see how SchemeUnit can provide direct support for them.

A more advanced approach is to generate tests from specifications (e.g. [6]). This approach naturally leads to model checkers like ACL2[19] and SPIN[15] that prove correctness. This is a powerful approach, though quite a leap from our simple system.

SchemeUnit only targets unit tests. In future we wish to target functional (whole system) testing, and testing of non-functional requirements such as performance. We are also aim to extend SchemeUnit to support domain specific functionality such as web site testing.

3. THE SCHEMEQL QUERY LANGUAGE

The International Standard Database Language[17] (SQL 1992, SQL'92 or just SQL) is a declarative language for manipulating data in database manager systems (DBMS). SQL is the standard interface to relational databases and is implemented by all major (and most minor) DBMSs. SchemeQL integrates a database manipulation language into the Scheme language offering an alternative to raw SQL.

Nowadays most database programmers already know SQL, and SchemeQL is designed to offer a gentle slope[16] from existing SQL knowledge to the higher level abstractions offered by SchemeQL.

We start by discussing the limitation of embedded SQL and why an alternative is desirable. We then describe the design and implementation of SchemeQL. We follow with an extended example that shows how SchemeQL builds on SQL but provides extended functionality that makes programming in SchemeQL easier than SQL. We finish with a discussion of related and future work.

3.1 The Limitations of Embedded SQL

The traditional approach to mixing SQL with another language is to embed the SQL as text strings. Even supposedly modern languages like Java [12] continue this tradition. The disadvantages of this approach are:

- SQL statements are not checked until execution time. It is easy to make grammatical or type errors when embedding SQL. For example, forgetting to include a space when concatenating two strings is a common error. Similarly one can write a SQL statement that uses SQL constructs where they aren't allowed, or uses the wrong type for arguments to SQL functions and so on. All these errors will cause execution time exceptions that may affect end users, whereas compilation time exceptions would have been caught and dealt with by the programmer.
- SQL statements can not be manipulated like host language statements. Except by using crude text processing one cannot programaticaly compose, abstract, and refine SQL statements. Hence code quality and programmer productivity suffer when using embedded SQL

If SQL statements were first class members of the programming language we could use our existing tools and language constructs to work with them, avoiding the problems given above.

3.2 The SchemeQL Design: a better SQL

SchemeQL embeds in Scheme a little language for creating and manipulating SQL queries. SchemeQL allows complex structured statements to be treated as first class citizens, thus considerably raising the level of abstraction a programmer can use.

The SchemeQL grammar is very *schemish* while following closely, in spirit, the SQL grammar. This eases the implementation as SQL is a complex mix between the relational algebra and the relational calculus, but more importantly allows the programmer to use their existing knowledge of basic SQL constructions and programming in Scheme. Furthermore, by making SchemeQL a set of syntactic extensions and procedures we can concentrate on the design of our little language, while retaining the whole power of a real programming language, Scheme, following the steps of other little languages [28], and [8].

SQL statements are divided into three main groups:

- Selection (SELECT)
- Modification (INSERT, UPDATE, and DELETE)
- Data definition (CREATE TABLE)

Selection (aka projection) statements produce a result set. Modification statements return a natural number representing the number of rows affected by the execution of the statement. Data definition statements are only interesting for their side effects, such as creating a new table or view in the database.

SchemeQL has the same logical division, with the following differences: result sets are represented by *cursors*, a lazy stream of rows (which basically allows the programmer to work with one row at the time), and instead of having a one to one mapping from SQL statements to Scheme procedures, we have a set of procedures to mimic the work of a single SQL statement. This simplifies the construction, combination, and refinement of statements. For instance, the full power of the SQL SELECT statement is achieved by the appropriate combination of several SchemeQL forms. Basic selection in SchemeQL follows this grammar:

```
selection    ::= (query <exp>)
              |  (query ((LITERAL <exp>)))
              |  (query <col-spec> <table-spec>)
              |  (query <col-spec> <table-spec>
                        <pred-spec>)
<exp>        ::= string-or-symbol
```

```
                  (passed verbatim to the DBMS)
<col-spec>    ::= ALL | (<column> ...)
<column>      ::= string-or-symbol | Number
                |  (<table> string-or-symbol)
                |  (AS <column> string-or-symbol)
                |  (LITERAL <exp>)
<table-spec> ::= <table>
                |  (<action> <table-spec> <table-spec>)
<table>       ::= string-or-symbol
<action>      ::= ALIAS | INNERJOIN | STRAIGHTJOIN
                | NATURALLEFTJOIN
<pred-spec>   ::= (<op> <col-spec> <col-spec-or-value>)
                |  ([AND|OR|NOT] <pred-spec> ...)
<op>          ::= < | <= | > | >= | = | <>
                | Any DBMS defined binary operator
<col-spec-or-value> ::= <col-spec>
                | Any value suitable for comparison
```

It is important to note, that the subforms in query, and in most forms in SchemeQL for that matter, are implicitly backquoted. Thus, (query ALL ,(f x)) means "select everything from the table, or tables returned by the application of Scheme procedure *f*, to the Scheme variable x".

3.2.1 More on Selection, and the SchemeQL Times

The query procedure alone does not provide all the functionality a programmer may want when selecting data from a database, and for a good reason: it would be as complex as the SQL's SELECT statement. Instead of offering a much too complex form, SchemeQL provides a set of forms, and procedures to specialized, compose, and otherwise handle selections. These forms are: query, distinct!, group-by!, order-by!, having!, limit!, union, intersect, and difference.

```
<selection> ::= (distinct! <selection>)
              |  (group-by! <selection> <limit-col>)
              |  ... the other forms
<limit-col> ::= ([ASC|DESC] <col-spec>) | <col-spec>
```

The syntax of the rest of the forms is just minor variations of that given above.

The reader may wonder what a SchemeQL selection exactly does. A selection in SchemeQL is an internal Scheme structure, that holds the information provided thus far to *perform* the selection, and that is why you can continue specializing it.

(query param ...) ⇒ query-struct

This is what we called the *SchemeQL compilation time*, for it allows us to perform basic static checking, based only on the information already provided to perform the selection. Only when schemeql-execute is called is the selection is performed and a result set (also called a *cursor* in SchemeQL) is returned.

(schemeql-execute schemeql-struct [conn]) ⇒ cursor

This is the *SchemeQL execution time*. The same scenario repeats itself for the data modification and data definition forms in SchemeQL.

Since sometimes we want to immediately execute a form, SchemeQL provides some useful shorthands for some forms that combine the generation of the internal structures and their execution. Here are some such forms that we will use later:

(direct-query conn param ...) ⇒ cursor
(query-with-current-connection param ...) ⇒ cursor
(query/cc param ...) ⇒ cursor

where conn is an open connection to a DBMS, which is created by a call to the SchemeQL form connect-to-database, and param ... are exactly those parameters valid for query.

3.2.2 SchemeQL Cursors

SQL result sets can be seen as tuples that form a table. SchemeQL cursors are pairs of values, (row promise), where row is a list representing the first tuple in the result set, and promise is a cursor holding a promise (that has to be forced) to return the rest of the tuples in the result set.

$$\begin{array}{rcl} \text{cursor} & \to & \text{row} \times \text{promise} \\ \text{row} & \to & \text{listof any} \end{array}$$

A library to work with cursors is provided as part of SchemeQL. Programmers most likely will use the following basic procedures to work with cursors:

- (cursor-car cursor): returns the first tuple in cursor.
- (cursor-cdr cursor): returns the rest of cursor, another cursor, similar to the original only that the *next* element, if any, is on the cursor-car position of the returned cursor.
- (cursor-null? cursor): *#t* iff cursor is the empty cursor.
- (cursor-map proc cursor): returns another cursor, whose first element is the application of proc to the first element in cursor, and whose second element holds the promise to apply proc to the rest of cursor.
- (cursor->list cursor N): returns a list containing the first N, or less if there are not enough, rows in cursor.
- (finite-cursor->list cursor): returns a list containing all the elements of cursor.

It is worth noting that cursors in SQL are a completely different concept, and are used to retrieve a small number of rows at a time out of a larger query. SchemeQL also provides support for them, through the procedures open-cursor, which receives a query and optional information to create different *kinds*[2] of cursors, the initial size of the set, and the starting row. Two other procedures work on the result of open-cursor: roll-cursor!, that changes the orientation of the given cursor, and close-cursor!, which closes the given cursor.

One important feature of this way of handling SQL cursors is that the resulting set of tuples is represented as an

[2]Kinds as those defined by Open Database Connectivity (ODBC) [27], which are: FORWARD ONLY, STATIC, KEYSET DRIVEN, and DYNAMIC.

SchemeQL cursor, and thus can be handled in the same way as the result of regular queries. We will not go into more detail here for space reasons.

3.2.3 The Rest of SQL

Most of the "usual" SQL functionality is already part of SchemeQL. Transactions, for instance, can be handled in two different ways. The first one, is by using the (transaction exp ...) form which executes all the expressions given in order, and if no exception occurs then it *commits* the block, otherwise, it sends a *rollback* to the DBMS, and pass along the exception. The transaction form tries to set the transaction isolation level to the highest possible, ideally to *serializable* level.

The second way allows the programmer to select the isolation level required and is represented by two procedures: begin-transaction and end-transaction. The begin-transaction form switches to manual commit mode, and sets the isolation level to the highest supported by the DBMS, or to the requested one if given. Then end-transaction either commits or rolls back the transaction block, depending on the argument supplied by the programmer. The transaction form is more scheme-like, since the other two can lead to the common error of opening a transaction, executing a block of expressions, and never closing the transaction again.

SchemeQL supports basic user and table management, and connection management that allows simultaneous connections to different databases. Even non-standard, yet very useful and regularly employed, SQL extensions such as CREATE DATABASE and USE DATABASE are supported, though no SQL standard procedure depends internally on these extensions.

3.3 The SchemeQL Implementation

SchemeQL is layered upon SrPersist[3] and takes full advantage of SrPersist's knowledge of the particular DBMS in use. SrPersist provides a safety check for every SQL statement sent to the DBMS, in addition to the SchemeQL's error detection, and thus we can offer a hierarchical approach to error handling.

SchemeQL together with SrPersist is a highly portable library since ODBC is the *de facto* standard for database connectivity and is widely supported (although it should be noted that many ODBC drivers have different levels of conformance[4]). In this regard SchemeQL offers two specific and crucial benefits. Firstly it hides the tedious and ugly details of the ODBC conformance levels from the Scheme programmer. Secondly, and more importantly, it removes the complexity of standard ODBC manipulation, which is probably the biggest drawback of ODBC when compared to other DBMS drivers.

[3]SrPersist is an ODBC library for PLT Scheme. More information on SrPersist can be found at:

http://www.plt-scheme.org/software/srpersist/

[4]At the time of writing there have been several major releases, from 1.0 through 3.51, and SrPersist supports them all.

Even though, for portability reasons, we use SrPersist, SchemeQL allows the use of different DBMS drivers. ODBC drivers are known to do extensive error checking, and so it is possible to have a database specific driver outperforming a generic ODBC driver. SQL support and basic error checking facilities are independent of the driver in use.

3.4 SchemeQL in action

All examples below are based around the following database structure. Suppose you own a software company, and the following tables are a snippet of your employees database.

personnel

id	name	lid
1	Noel	1
2	Ian	1
3	Francisco	1
4	Simon	2
5	James	3
6	Brian	4
7	Dennis	4

salaries

id	salary
1	30'000
2	30'000
3	30'000
4	30'000
5	45'000
6	45'000
7	45'000

languages

id	lang
1	Scheme
2	Haskell
3	Java
4	C

We start with the most common sort of query, which is a SELECT statement such as the following statement to get the names of all the programmers:

```
SELECT name FROM personnel
```

In SchemeQL this query has almost exactly the same structure as its SQL equivalent:

```
(query (name) personnel)
```

Now suppose we wish to get all the *id*s of those employees who program in Scheme. In SQL we'd write:

```
SELECT personnel.id
  FROM  personnel, languages
  WHERE personnel.lid = languages.id
  AND   languages.lang = 'Scheme'
```

In SchemeQL we write

```
(query ((personnel id))
       (personnel languages)
       ((= (personnel lid) (languages id))
        (= (languages lang) "'Scheme'")))
```

Again the two queries have a very similar structure. Now suppose we want to get all Java programmers. Immediately we see an opportunity for code reuse if we parameterize the above queries on the language. This is trivial in SchemeQL as we can use abstraction facilities provided by Scheme:

```
(define (programmers language)
  (query ((personnel id))
         (personnel languages)
         ((= (personnel lid) (languages id))
          (= (languages lang) ,language))))
```

Remember that most subforms in SchemeQL are backquoted.

There is no way to do this in standard SQL, though individual DBMSs may provide parameterized queries. To do this in embedded SQL we could append strings:

```
(define (programmers language)
  (string-append
    "SELECT id "
    "FROM personnel, languages"
    " WHERE personnel.lid = languages.lid "
    "AND   languages.lang = " language))
```

We note that this method is error-prone as it is easy, for example, to forget to include a space between strings as we have done above (between *languages*, and the keyword *WHERE*).

Now suppose you want to get the *id*s of all C programmers who are earning 45'000. This is the intersection of all C programmers, which we already know how to do, with all programmers who are earning 45'000. In SQL we can write:

```
SELECT id
  FROM personnel, languages
  WHERE personnel.lid = languages.lid
  AND   languages.lang = 'C'
  INTERSECT ( SELECT id
              FROM salaries
              WHERE salary = '45000' )
```

In SchemeQL we can form the two sets separately and then perform the intersection:

```
(let ((c-programmers (programmers "'C'"))
      (high-earners (query (id) (salaries) (= salary "45000"))))
  (intersect c-programmers high-earners))
```

Notice how we have reused the programmers function defined above and then composed a query from parts. We cannot do this in SQL.

That does it! Impressed by the productivity of your functional programmers you decide to fire all the Java and C programmers and use the extra money to give a raise to your fine Scheme programmers (you find the Haskell programmers productive but inexplicably lazy). Coincidentally this also give us an opportunity to show further query composition and cursor handling in SchemeQL.

First we define the sets of interest: the Schemers, who are getting a raise, the Haskell programmers, who just stay as they are, and everyone else, who are getting the opportunity to explore other interests.

```
(define schemers (programmers "'Scheme'"))
(define haskellers (programmers "'Haskell'"))
```

```
(define fired
  (let ((all (query (id) personnel)))
    (schemeql-execute
      (difference all (union schemers haskellers)))))
```

Now all programmer who have been fired are removed from the salaries table:

```
(cursor-map
 (lambda (programmer)
   (let ((id (car programmer)))
     (delete/cc salaries (= id ,id))))
 (result-cursor fired))
```

Finally, to give the Scheme programmers a raise:

```
(cursor-map
 (lambda (id)
   (update/cc salaries
              ((salary (LITERAL "salary * 2")))
              (= id ,(car id))))
 (result-cursor (schemeql-execute schemers)))
```

The above operations cannot be performed in pure SQL as query results cannot be used as the input to modification statements. We give below equivalent statements to perform the above actions. Where an action requires repetition of a number of very similar statements (eg, when DELETEing the imperative programmers) we only give an example.

```
SELECT personnel.id
   FROM personnel
   EXCEPT (SELECT personnel.id
            FROM personnel,languages
            WHERE personnel.lid = languages.lid
                  AND languages.lang = 'Scheme'
           UNION
           SELECT personnel.id
            FROM personnel,languages
            WHERE personnel.lid = languages.lid
                  AND languages.lang = 'Haskell');
DELETE FROM salaries
   WHERE id = 4;
SELECT personnel.id
   FROM personnel
   INTERSECT (SELECT personnel.id
            FROM personnel,languages
            WHERE personnel.lid = languages.lid
                  AND languages.lang = 'Scheme');
UPDATE salaries
   SET salary = salary * 2
   WHERE id = 1;
```

3.5 Related and Future Work

Haskell/DB, a compiler embedded in Haskell that dynamically generates SQL queries, was developed as an instance of the more general design pattern for embedding client-server style services into Haskell detailed in[22]. Some of the benefits this technique offers are:

- Programmers need to know only one language,

- it allows language extensions in the form of libraries to be presented,
- it is possible to impose specific typing rules,
- integration with other domain specific libraries (e.g. CGI, mail) is possible, and finally
- this approach offers a strategic advantage, for it empowers programmers to use the language infrastructure, such as the module, and type systems.

SchemeQL has all of these benefits except for static typing.

The implementation of Haskell/DB, presented in[22] uses ActiveX Data Objects (ADO) to connect to the DBMS. In this regard Haskell/DB is limited to the Windows platform. SchemeQL does not share this limitation as it uses SrPersist, which can interact with any ODBC driver.

Our approach is to define a limited domain specific language that can be translated into SQL. Another approach is to expand the database query language into a full programming language[29]. This approach has many benefits but requires the underlying DBMS to change.

It is clear that *structured data* is taking over. The Extensible Markup Language [26] (XML) is now considered the universal format for structured documents and data on the Web. With XML arises the need for efficient query languages to exploit structured data. XML Query [24] is a working group aiming to create a set of query facilities to extract data from XML, or viewing XML files as databases. Unfortunately there is not yet any direct point of comparison between XML and current database technology. This will remain one of the most interesting topics of research in the years to come. Whether or not a language like SchemeQL will be able to enter the XML realm is a question we cannot answer yet.

In the immediate future we will be adding support for specific DBMS drivers and SQL dialects (e.g. Oracle, PostgreSQL, etc.). We will also attempt to standardize the SchemeQL syntax as a Scheme Request for Implementation[21] (SRFI).

4. SCHEMEUNIT AND SCHEMEQL

SchemeUnit and SchemeQL have both been designed with a 'gentle-slope' philosophy: start with an already familiar base and then build additional functionality as independent components on that base. In SchemeUnit this is evident in the way test code mimics the "code a little, test a little" cycle and adds facilities to organize and rerun tests. In SchemeQL the starting point is the SQL `SELECT` statement upon which the `query` macro is modeled. The combinators `intersect`, `difference` and so on are then introduced as ways of modifying the basic `query`.

SchemeUnit and SchemeQL both take advantage of Scheme's macro facilities to present a cleaner interface to the user. In both languages macros are used to avoid repetitious `lambda` statements. In SchemeUnit this is in the creation of test cases. In SchemeQL this is in cursor creation. Macros are also used for other purposes: in SchemeUnit to allow user-extensions via the define-assertion macro and in SchemeQL to provide implicit backquoting on forms. These simple uses of macros go a long way to improving the user experience.

SchemeUnit is used extensively to test itself and SchemeQL.

5. CONCLUSIONS

A language is a user interface just like a graphical interface and deserves as much attention from the language designer as a GUI would get from it's designer.

We have described SchemeUnit, a little language for writing tests in Scheme, and have illustrated how we have used the features of functional languages in general, and Scheme in particular, to simplify the interface. Via comparison with the "code a little, test a little" cycle and the JUnit framework we have shown that SchemeUnit achieves an admirable level of simplicity without sacrificing expressive power.

SchemeQL, our little language for database interaction, has been shown to be a feasible alternative to embedded SQL. By building on the programmer's knowledge of SQL and extending it with modular combinators we achieve tighter integration with the Scheme language, a better, more modular, parameterization of SQL statements and improved expressibility and abstraction.

PLT Scheme, the host language for both our little languages gives us a certain number of extra, and free advantages that makes them usable, through its DrScheme programming environment[11]: a syntax-sensitive editor, a syntax checker, an stepper, and interaction with other libraries, and plugins.

- Since our little languages consists entirely of tree-structure expressions, the editor's features are inherited. Users only needs to add the keywords in our little languages to DrScheme (to have them indented appropriately.)
- No modification is needed to work with the syntax checker, and the stepper since these two work transparently over procedures, and macros.
- Since all of the host language is available to users, a program can load, or enable a certain number of libraries, plugins, or other embedded little languages as needed with no extra fuss.

The only extra advantage we are not exploiting is the validity checking available through the MrFlow component of DrScheme though it should not be hard to expand the constructions of our little languages to type definitions, as in [8].

Finally we note that our language evaluation has been qualitative; based on our experiences using the languages in question. We are aware of some work in quantitative evaluation[7] and this research will contribute to a better understanding of what makes good language design.

6. ACKNOWLEDGMENTS

We are indebted to the following individuals:

Ryan Culpepper, who created the graphical interface to SchemeUnit and has contributed greatly to its design.

Paul Steckler, Shriram Krishnamurthi, Matt Jadud, MJ Ray and the anonymous reviewers who offered comments on the draft versions of this paper.

7. REFERENCES

[1] F. A. Adrian. Clunit. http://www.ancar.org/CLUnit/docs/CLUnit.html, 2002.

[2] K. Beck. *Kent Beck's Guide to Better Smalltalk*, chapter 21. SIGS Reference Library. Cambridge University Press, 1999. http://www.xprogramming.com/testfram.htm.

[3] K. Beck. *Extreme Programming Explained.* Addison-Wesley, 2000.

[4] K. Beck and E. Gamma. Test infected: Programmers love writing tests. *Java Report*, 3(7), July 1998.

[5] J. Beekmann. Curlunit. http://curlunit.sourceforge.net/, 2002.

[6] Y. Cheon and G. T. Leavens. A simple and practical approach to unit testing: The jml and junit way. Technical Report 01-12, Department of Computer Science, Iowa State University, November 2001.

[7] S. Clarke. Evaluating a new programming language. In G. Kadoda, editor, *Proceeding of the 13th Workshop of the Psychology of Programming Interest Group*, volume 13, April 2001.

[8] J. Clements, P. Graunke, S. Krishnamurthi, and M. Felleisen. Little languages and their programming environments. In *Proceedings of the Monterey Workshop*, 2001.

[9] P. Doane. Fort: Framework for o'caml regression testing. http://www.sourceforge.net/projects/fort, 2002.

[10] M. Feathers. The 'self'-shunt unit testing pattern. http://www.objectmentor.com/-resources/articles/SelfShunPtrn.pdf, 2001.

[11] R. Findler, J. Clements, C. Flanagan, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: A programming environment for scheme. *Journal of Functional Programming*, 2001.

[12] M. Fisher, R. Cattell, G. Hamilton, S. White, and M. Hapner. *JDBC API, Tutorial, and Reference, Second Edition: Universal Data Access for the Java 2 Platform*. The Java Series. Addison-Wesley Longman, 1999.

[13] P. Graham. *On Lisp*. Prentice Hall, 1993.

[14] D. Herington. Hunit. http://hunit.sourceforge.net/, 2002.

[15] G. J. Holzmann. The model checker spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[16] M. Hostetter, D. Kranz, C. Seed, C. Terman, and S. Ward. Curl: A gentle slope language for the web. *World Wide Web Journal*, II(2), 1997. http://www.w3j.com/6/.

[17] Database language sql. International Organisation for Standardization (ISO), 1992.

[18] R. Jefferies. Software downloads. http://www.xprogramming.com/software.html.

[19] M. Kaufmann, P. Manolios, and J. S. Moore, editors. *Computer-Aided Reasoning: ACL2 Case Studies.* Kluwer Academic Publishers, June 2000.

[20] G. King. Lift - the lisp framework for testing. Technical report, University of Massachusetts, 2001.

[21] S. Krishnamurthi, D. Mason, and M. Sperber. Scheme request for implementation. http://srfi.schemers.org/, 1998.

[22] D. Leijen and E. Meijer. Domain specific embedded compilers. In *2nd USENIX Conference on Domain-Specific Languages (DSL)*. USENIX, October 1999.

[23] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: Unit testing with mock objects. In G. Succi and M. Marchesi, editors, *Extreme Programming Examined*. Addison-Wesley, 2001.

[24] M. Marchiori. Xml query. http://www.w3.org/XML/Query, 2000.

[25] M. T. Nygard and T. Karsjens. Test infect your enterprise javabeans. *JavaWorld*, May 2000.

[26] L. Quin. Extensible markup language (xml). http://www.w3.org/XML/, 1997.

[27] R. E. Sanders. *ODBC 3.5 developer's guide.* McGraw-Hill, 1998.

[28] O. Shivers. A universal scripting framework, or lambda: The ultimate "little language". In J. Jaffar and R. H. C. Yap, editors, *Concurrency and Parallelism: Programming, Networking, and Security*, volume 1179 of *Lecture Notes in Computer Science*, pages 254–265. Springer, 1996.

[29] V. Tennen, P. Buneman, and S. Naqvi. Structural recursion as a query language. In *Proceedings of 3rd International Workshop on Database Programming Languages*, 1991. http://db.cis.upenn.edu/Publications/.