The following paper was originally published in the
Proceedings of the Fourth USENIX Tcl/Tk Workshop
Monterey, CA, July 10-13, 1996

QuaSR: A Large-Scale Automated, Distributed Testing Environment

Steven Grady, G. S. Madhusudan, and Marc Sugiyama
Sybase, Inc.

QuaSR: A Large-Scale Automated, Distributed Testing Environment

Steven Grady, G. S. Madhusudan, and Marc Sugiyama
(steveng@sybase.com, madhu@sybase.com, sugiyama@sybase.com)

Sybase, Inc.

Abstract

The *QuaSR* project at Sybase, Inc. involves the creation of thousands of automated tests for Sybase, Inc.'s *SQL Server*, each implemented as an independent Tcl program. The resulting test suite, significantly more than a million lines of code, comprises the largest known Tcl code base. The test harness is written in [incr Tcl] and the test cases in Tcl. Tcl and [incr Tcl]'s extensibility, simplicity, and reliability have made them uniquely suited to the development of a sophisticated automated testing system.

1. Introduction

QuaSR (Quality Systems Re-engineering, pronounced "quasar") is an internal project at Sybase, Inc. to improve the quality of its software, starting with its relational database server product, *SQL Server*. The primary technical component of this effort is the *QuaSR* test harness. The harness provides an environment in which small automated test programs may be developed to test arbitrary pieces of functionality, and sets of tests may be combined into single test runs, with distributed resource allocation being handled automatically.

This paper provides an overview of the design of the QuaSR test harness (also known as "QuaSR"), along with samples illuminating the use of the system. It concludes with lessons learned from the application of Tcl and [incr Tcl] to an automated testing environment such as ours.

2. Test Harness Design

2.1 Overview

The goal of the QuaSR project is to deliver a fast, reliable, extensible, and automated test system. Specifically, it must enable overnight execution of the main body of SQL Server regression tests. To this end, it provides for:

- minimized, automated resource allocation
- self-analyzing, assertion-based test cases
- distributed client/server execution
- standardized test components
- test case independence

2.2 Test Case Files

(To better understand the system, the following discussion will refer to the example Test Case File, `rtrim.tcf`. For the sake of brevity, the free-text sections have been edited.)

A *Test Case File*, or *TCF* (pronounced "tee-kiff"), is a file containing a set of test cases. The test cases are usually related in the functionality they test. The TCF consists of a declaration of resources available to all test cases, an initialization routine, called *tcf_start*, a termination routine, called *tcf_end*, and a set of one or more test cases.

In *rtrim.tcf*, one resource is declared, a standard database of type *stdbempty* (a database with a full complement of users defined, but no additional data). It is given the logical name *mydb*. In *tcf_start*, a database connection is established to the SQL Server on which *mydb* resides. The connection is established through the user *loginA* and given the name *sqlcon*. This connection is established before any test case in the TCF is executed. *Tcf_end* closes the *sqlcon* connection; it is invoked after all test cases are complete. The `tcf_run` line at the end is an imple-

```
resources {
      stdbempty mydb
}

tcf_start {
      mydb login loginA sqlcon
}

tcf_end {
      sqlcon close
}

testcase 1 -assertion {
      When "rtrim" is used with less than or greater than one argument, then
      error 174 is generated.
} -strategy {
      1. set up list of no argument and two arguments
      2. execute the following command:
            select rtrim(argumentlist)
      3. if SQL Server does not return error number 174 return FAIL
      4. repeat steps 2 and 3 for all test variations
      5. if all test variations are successful, return PASS,
            otherwise, generate UNRESOLVED
} -code {
      set pass_count 0
      set testlist {
            { }
            { abc, xyz }
      }
      set totvariation [llength $testlist]
      foreach test $testlist {
            set cmd "select rtrim($test)"
            SQL_cmd RESULT sqlcon $cmd
            if { ! [$RESULT servermsg 174] } {
                  util_log "expected error 174 NOT returned by server"
                  return FAIL
            }
            incr pass_count
      }

      if { $pass_count == $totvariation } {
            return PASS
      } else {
            error "Expected variations $totvariation but got $pass_count"
      }
}

tcf_run
```

rtrim.tcf

mentation artifact.

The *resources* section is simple and powerful. In *rtrim.tcf*, it contains only the declaration of a single database. However, QuaSR automatically supplies any additional resources necessary to support a standard database; specifically it results in the implicit declaration of a SQL Server to host the database, three logical devices to support the SQL Server and the database, three physical devices (potentially raw devices or filesystem space) to support the logical devices, a machine that holds the devices and runs SQL Server, and a network address for client-server communication.

Resources can also be declared more explicitly if non-standard configurations are required. For instance, a standard database could also be declared more explicitly as:

```
machine mymac
sql_server -machine mymac mysrv
stdbempty -sql_server mysrv mydb
```

The above declaration would ultimately result in exactly the same resource allocations as the simple declaration used in *rtrim.tcf*. The declaration syntax can be used to declare arbitrarily complex resource relationships, however. One can declare resources for tests that require interactions between multiple databases, SQL Servers, or machines, non-default device types and sizes, specific platforms or localization values, etc.

2.3 Test Cases

A *test case* is the smallest unit of functionality in QuaSR. Each test case is given an identifying number and has text describing the assertion under test, a strategy providing an English description of the approach used for the test, and a program implementing the test. The first two sections are normally logged when a test case fails so that all relevant information is readily available for test failure analysis.

rtrim.tcf contains one test case, which tests a negative assertion (one expected to generate an error). Using the resources declared in the *resources* section and the connection created in *tcf_start*, it creates the appropriate SQL commands to test the assertion, sends them through the connection, and analyzes the results. Depending on the results, the test case may generate a PASS, indicating the assertion was found to be valid, a FAIL, indicating the assertion was determined to be invalid, or an UNRESOLVED (caused by the `error` call), indicating that there was some problem with the test case and that the assertion could not be tested.

2.4 Test Sets

A test set is a set of test cases. An example test set is:

```
tests/dml/rtrim.tcf
tests/dml/select.tcf{1,3,5-8}
tests/ddl
```

This test set specifies a set consisting of all the test cases in *rtrim.tcf*, a subset of the cases in *select.tcf*, and all the test cases in the directory hierarchy *ddl/*. Through other QuaSR mechanisms, it is possible to specify test sets such as “all tests that do not require a tape drive”, “tests that require *SQL Server* version 11 that run only on HPs”, or “tests that exercise code in `cache.c`”.

A test set is used as the basis for resource allocation and usually as the basis for an execution session.

2.5 Test Sessions

A test session normally consists of the following steps:

- generate
- scan_res
- acquire
- prepare
- exec
- release

The steps *generate* through *prepare* are used to set up the environment to run a set of tests. The *exec* step runs the tests. The *release* step cleans up the session, allowing any acquired resources to be used by others.

Generate is responsible for converting a test set specification file into a format appropriate for use by the rest of the system. For instance, it traverses any directory hierarchies to determine the individual TCFs within them.

Scan_res scans the resource requirements of the TCFs. It reads each TCF, determines the full resource requirements of each, then consolidates the full set of TCF-declared resources into a *minimal resource set*. Thus, for instance, a set of 100 TCFs may require only a single SQL Server, on a machine with enough space for two standard databases. (Theoretically, the resulting set may not actually be minimal, but in practice, our algorithm almost always generates a minimal set.)

Acquire uses the minimal resource set as a basis for requesting a machine (or machines) that can support the resources. The resources are allocated from the *Resource*

Manager, and are locked for use during the test session. By allocating the complete set of resources before execution, the user is guaranteed that the test run will not fail during execution due to lack of resources. Platform type, operating system, SQL Server version, and other attributes of the resulting execution environment are recorded for later phases.

Prepare prepares the resources. Support directories are created on the machines, SQL Servers are initialized, databases are created. This step is the first in which the machines responsible for executing the tests are actually used.

Exec executes the test cases. The user has the option of specifying an execution scenario that contains a subset of those in the original test set. Each TCF is run in turn, binding the logical resource names to the acquired resources, running the *tcf_start* initialization, executing each of the test cases specified in the scenario, then cleaning up in the *tcf_end* routine. The results are logged in a journal file, with summary information of the run printed on the standard output. The journal file is a structured file containing all information relevant to a particular run, including resource attributes, debugging information for non-PASSing tests, and timing information.

Release releases the acquired resources, freeing them for others to use.

The separation of the default process into individual steps allows for greater control by the user. For instance, a SQL Server developer can prepare all resources, run the tests, fix bugs demonstrated by the test run, then (using other steps not described above) copy a new version of the SQL Server binary to the acquired machines to rerun the tests. A test developer could modify the *resources* declaration in a TCF, then re-scan the resources to verify that the currently-acquired resources are sufficient to support the new declaration. There are also compound steps, such as "setup", which are responsible for executing multiple basic steps.

2.6 User Configuration

QuaSR allows the test runner to specify certain resource configuration values. For instance, the user may specify that the unspecified machine platforms should default to "Sparc" or that physical devices should default to use raw partitions. Another configuration variable allows the user to specify an alternative binary to execute in place of the standard SQL Server (it will be copied automatically to the remote machines before execution), or to allow the user to run binaries under a debugger rather than invoking them automatically.

2.7 Graphical User Interfaces

There are multiple graphical interfaces to the system. One provides an interface to the steps described above (along with information about the current state of the system, buttons to provide terminal connections to SQL Servers, and other conveniences). Another provides a simple way to browse a journal file. Among other things, it provides buttons to traverse the hierarchical format in an intuitive fashion, uses multiple colors to distinguish different types of information, formats query results into a familiar format, and allows the user to inspect the original TCFs.

2.8 Test Code Support

Various support libraries and extensions allow test case writers to write code at a high level conforming to that of the strategy. *QASQL* provides commands to communicate with *SQL Server*, formatting the resulting data stream into a manipulable format. *Undo* provides a mechanism for expressing an undo stack; this mechanism is used to record any changes that a test case makes, so that after the execution of the test case, the SQL Servers can be restored to the same state as before the test case was run.

The *Utility Library* provides a set of common high-level procedures and objects to simplify the coding of standard test steps (for instance, the `RESULT` object in the example test case understands requests for specific pieces of information, such as whether a given server message was contained in the result). The *Log* library provides a way to generate uniform, parsable debug messages suitable for later filtering.

The *Resources* module, which defines the resources available for declaration, supplies a variety of additional methods for information retrieval and resource control, such as determining the exact size of an allocated device, or shutting down and rebooting a SQL Server.

2.9 Utility Scripts

In addition to the journal browser, there are various tools available for analyzing the results of a single run or set of runs. There are tools to process the journal and place the results in a database suitable for querying. There are scripts to summarize the results of a single run and for comparing two runs. *Mrsummary* summarizes the resource requirements for a particular session. *Showsql* shows the SQL commands that were sent to SQL Servers in a given session, optionally in a format suitable for input to *isql*, an interactive SQL shell.

2.10 Other Modules

The *agent* is a program that must run on each remote machine; it services requests to start processes and capture their output, create and remove files and directories, and other simple, operating-system-specific activities. It is the only part of the system which is ported to all the SQL Server platforms (which include, along with over a dozen UNIX variants, diverse platforms such as VMS, NT, OS/2, and NetWare). The *agentlib* library provides commands for the client to communicate with the server agents.

The *assertion database* stores information about assertions and their associated test cases. The *results database* stores the results of test runs (based on the contents of journal files).

Interactions in the system are controlled by a *state machine*, which determines what steps are legal at a given stage. This information enhances the main GUI, by limiting users to legal actions.

Tcl, [incr Tcl], and the various extensions are combined into a single interpreter, called *squash* (Sybase QUality Assurance SHell). *Squash* is the program which actually runs the code in the test cases.

There is an option to run a full test set in parallel. The algorithm used is very simple: to run *n* parallel threads, the tests are examined, and those requiring only a standard empty database are placed in *n-1* homogeneous buckets; the rest in one heterogeneous bucket. The buckets are then separated into test runs and run individually. Currently, about two thirds of the TCFs fit into the homogeneous buckets. Ignoring time requirements for TCFs, on average a suite can be split into three threads to parallelize at maximum effectiveness.

3. *SQL Server* Test Suite

As of the time of writing (March 1996), thousands of individual test cases have been coded for testing SQL Server, comprising a Tcl code base with a line count in the millions – the largest known Tcl code base in a single project[1]. Some of the cases test multiple variations; the total suite currently performs tens of thousands of product tests. Test suites for products other than SQL Server are also in development.

The SQL Server test cases vary in length from around 10 lines to thousands of lines, with the majority of the simpler cases being under 100 lines. They vary in complexity from those that create and execute a single SQL command and verify the result (as in the test case in *rtrim.tcf*) to those that have several nested loops, and check the results of tens of SQL commands.

Individual test cases run in times ranging from under one second to about twenty minutes, with the vast majority running in under five seconds (depending on the server platform). On a fast server (e.g. an HP9000/800G with 128 megabytes of memory), a single-threaded run takes under twelve hours.

4. Use of Tcl and [incr Tcl]

The QuaSR test harness is implemented as a combination of C, C++, Perl, Bourne shell, Tcl, and [incr Tcl]. C and C++ are used primarily to provide Tcl extensions (including QASQL, the debug log library, and the agent library) that implement Tcl APIs on top of existing C APIs. Perl and sh are used for a few of the utility scripts, primarily performing file manipulation and process control. The bulk of the design is in Tcl (version 7.3) and [incr Tcl] (version 1.5). The GUIs are written using Tk (version 3.6).

4.1 Statistics

There are about 17,000 lines of Tcl/[incr Tcl] in the harness (plus about 8,000 lines more for the data that go into the standard databases), and 10,000 lines of C and C++. (There are also about 1,000 lines of Bourne shell and Perl scripts).

4.2 [incr Tcl] classes

The primary use of [incr Tcl] is in the definition of resources. Each resource is a separate class, organized into a single-inheritance hierarchy of about a dozen classes in total. Another hierarchy is used to define types of *Resource sets*, including the resources associated with a single TCF and the minimal resource set. Class containment is used to describe the relationship between platform-specific versions of SQL Server.

Other [incr Tcl] uses include: the utility library, which uses objects to create high-level interfaces to result streams coming back from SQL queries, and the resource manager, which defines machine attribute requests as objects.

1. This claim is based on responses to a query posted to comp.lang.tcl in February 1996.

4.3 Conceptual Expression

Nearly all of the conceptually challenging parts of QuaSR are written in Tcl/[incr Tcl]. The resource minimization algorithm, the dynamic resource binding, and the parallelization algorithm are coded in Tcl. The only uses of C/C++ are for C API module interfaces and for the journal browser. The browser must be able to parse and display files of 10 megabytes and more, so the processing code was rewritten in highly optimized C.

4.4 Test Case Tcl Usage

Because few of the test developers on the project had experience in object-oriented design and implementation, we decided that they could ramp up more quickly if they did not have to learn about [incr Tcl] and class design. In test case code, we minimize the exposure of [incr Tcl] to the use of method invocations on declared resources within the test cases. (Of course, we make full use of [incr Tcl]'s capabilities in the test harness code.)

5. Benefits of Tcl and [incr Tcl]

There was some initial apprehension about the choice of Tcl and [incr Tcl]. In hindsight, Tcl and [incr Tcl] have been suitable for both the harness and the test suite.

5.1 Easy to Learn

Part of the project was hiring and training a group of programmers who would develop tests under our system. At the time of hiring, almost none of them had used Tcl. Tcl's simplicity reduced the ramp-up time for developing tests under QuaSR.

5.2 Extensible

The use of [incr Tcl] classes as the basis for resource design made possible a truly powerful resource declaration language that can easily be extended as new resource requirements are identified. Even significant redesign of the class hierarchy has been possible due to the modular coding available with [incr Tcl]. During the course of the project, new resource classes have been added and old ones changed, with no backwards incompatibilities in the TCFs and few changes in other parts of the harness.

5.3 Embeddable

Standing alone, Tcl with [incr Tcl] would not have been sufficiently powerful to meet the goals of QuaSR in a reasonable timeframe. The embeddable nature of the Tcl interpreter made it possible to build a superstructure (*squash*) that defines procedures and objects at an appropriate conceptual level.

5.4 Interpreted

Because Tcl is interpreted, and because of the easy transformation between code and data, it has been possible to implement some very powerful constructs around the test cases. Test case code is stored as an [incr Tcl] instance variable in an object, and later executed. The execution is wrapped in code that catches exceptions, invokes user-specified hooks, and processes *undo* statements to reset the environment, all implemented with almost no effort. Adding further functionality, such as invariant checks between test cases, is similarly simple.

The interpreted nature of QuaSR has also made it easy to test the harness itself, since the tests have easy access to the internal harness procedures. Also, it was possible to implement simple tests for the GUIs by invoking widget actions via Tk's `send` command.

5.5 Public Domain

Because most of the components of QuaSR are based on public-domain code (Tcl, [incr Tcl], Don Libes' Tcl debugger, Perl, etc.), experimentation with different potential tools required little investment of time or money. The reduced risk made it possible to take more chances when trying to find the right solution.

The reliability of both Tcl and [incr Tcl] has been outstanding, attributable at least in part to their public-domain status, resulting in thousands of developers being able to identify and fix problems.

5.6 List and String Processing

Ultimately, testing SQL Server comes down to sending *Transact-SQL* commands to the server and verifying that the results are as expected. A language that allows strings to be created easily is appropriate to generating the commands to be sent, and a language that handles structured lists easily is appropriate for analyzing the response stream. The resulting test case code has been easy to read and write.

5.7 [incr Tcl] Cleanly Integrated

Minimizing the exposure of [incr Tcl] to the test case writers would not have been possible had [incr Tcl] not been cleanly designed. As it is, test coders do not need to learn anything about [incr Tcl] beyond the structure of a method invocation.

6. Disadvantages of Tcl and [incr Tcl]

While most of our initial concerns about the use of Tcl and [incr Tcl] were unfounded, there are a few areas that need to be improved before their use in this project can be called a complete success.

6.1 Performance

Although the bulk of the processing during test case *execution* is on the server, much of the *preparation* time is on the client, particularly the *scan_res* phase which is responsible for consolidating the resources. The bottleneck here is [incr Tcl]'s performance when handling more than a handful of objects. [incr Tcl] 2.0 is supposed to alleviate this problem but we have not completed the transition to [incr Tcl] 2.0.

The time for simply parsing the TCFs is non-trivial when they comprise over a million lines, and complex algorithms can be painfully slow.

6.2 Memory Consumption

[incr Tcl] 1.5 is a memory hog. This becomes a problem when a large amount of data is held in memory. Again we are hoping [incr Tcl] 2.0 will alleviate this problem.

6.3 Lack of Development Tools

Although we make some use of public-domain development tools, such as Don Libes' debugger, we sorely feel the lack of industrial-strength tools,

- a truly integrated debugger that also understands [incr Tcl].
- a profiler with profiling for Tcl procedures, and [incr Tcl] objects and methods.
- a syntax checker.
- code coverage analyzer. In particular, each test case must be visually inspected to make sure there are no syntax errors, a tedious effort that a compiler would render unnecessary.
- Tcl compiler - we are testing a couple of recently released compilers.
- object browser for [incr Tcl]. This is mandatory if any serious programming has to be done in [incr Tcl]. Nautilus is a start, but we need tools similar to those suppled with most advanced C++ environments.

6.4 Rapid Change in Tcl Versions

QuaSR is an extremely large system which is rolled out to 300-500+ users currently. The frequency with which Tcl and its extensions get changed is a double edged sword. On one hand we get quick bug fixes, but the price we pay is frequent upgrades. Once the system goes production, frequent upgrades will not be feasible.

6.5 Error Reporting

Error reporting can be improved. The problem is mainly with the specificity of errors with respect to location and the nature of the problem.

7. Use of Alternate Test Harnesses

QuaSR uses the X/Open TET harness as its underlying harness. It was chosen primarily because of its successful use in other projects and its support for assertion based tests. Its distributed nature was also a factor in its selection. DejaGnu could also have been used but we were not sure about its maturity and ability to support distributed tests. Having said this, it should be pointed out that the test harness plays a very minor role in the current QuaSR system and hence the choice of a test harness is not very germane. This may change if the system is used for interactive testing, where DejaGnu has definite advantages. But in the current non-interactive regime of tests, most of the work is done by the test case and the distributed agent with the harness merely acting as a test case sequencer.

If we ever decide to switch to DejaGnu, the effort will not be significant since the underlying harness is fairly well isolated from the rest of the system.

8. Futures

QuaSR is undergoing continuing development. The following changes are expected in the near future:

8.1 New Features

QuaSR was originally designed for SQL Server testing. Relatively simple mechanisms can be added to support client testing as well, including interoperability testing of arbitrary client-server combinations.

Support for testing server products other than *SQL Server* can be added relatively easily through additions to the resource class tree. Once the new resources are in place, tests can simply declare the new resource and use it. The new tests would operate cleanly with tests in the existing suite.

8.2 Updated Software

QuaSR is currently being rolled out to various test and development groups at Sybase, Inc. In order to preserve stability, we have not integrated the latest versions of the underlying software. We expect to switch to Tcl 7.5, Tk 4.1, and [incr Tcl] 2.0 when we have the time to deal with any problems generated by the switch.

8.3 Performance Optimization

The current system meets the broad performance goals. However more work is required before all the specific performance goals are met.

9. Conclusions

The QuaSR project is ambitious both in its scope and its performance goals. The use of Tcl and [incr Tcl] as a basis for its implementation has its pros and cons but the benefits outweigh the disadvantages. The only significant drawback is the lack of industrial-strength development tools.

From a development standpoint, it is clear that the high-level nature of Tcl was beneficial. Both the harness design team and the test writing team found that the bulk of the design time was spent thinking about conceptual problems; once a solution was devised, it was straightforward to implement in Tcl. The developers spent their time much more effectively than had they been using C or another low-level language.

As a testing tool, Tcl's clean syntax and easy manipulation of structured data allow for powerful tests to be written simply and clearly. Inefficiencies due to interpretation are of little importance, particularly given the client-server nature of the product under test. Similarly, the high level interfaces made convenient by [incr Tcl] allow for powerful support libraries. High-level libraries allow tests to be written at a level close to that of their pseudo-code strategies.

Although there are challenges involved in using Tcl for a single, large-scale program, it is perfectly suited to the development of (thousands of) small programs. In particular, QuaSR, despite its size, has not been impaired by the use of Tcl, and in fact has been well-served by Tcl and [incr Tcl]'s reliability, simplicity, and extensibility.

10. Bibliography

[Libe] Don Libes, "A Debugger for Tcl Applications", *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[McLe93] M. J. McLennan, "[incr Tcl]: Object-Oriented Programming in Tcl", *Proceedings of the Tcl/Tk Workshop*, University of California at Berkeley, June 10-11, 1993.

[Savo] Rob Savoye, "The DejaGnu Testing Framework". Available at http://www.cygnus.com/doc/dejagnu/dejagnu_toc.html.