ROSE Matrix Testing

Terminology

Matrix testing is the process of testing a single version of ROSE with a number of configurations of software dependencies and other configuration settings (together called “dependencies”). This creates an *n*-dimensional space that defines where ROSE works and doesn't work, where *n* is the number of dependencies. A coordinate in the *n*-dimensional is indicated by an ordered *n*-tuple called a configuration vector. Since the configuration vectors are too long for humans to read easily, we always write them as an unordered list of *key=value* pairs, like this:

```
assertions=abort boost=1.54 build=autoconf compiler=llvm-3.5 debug=yes dlib=none doxygen=1.8.9 edg=4.7
languages=binaries magic=none optimize=no python=/usr/bin/python3 qt=none readline=none sqlite=none
warnings=yes wt=none yaml=0.5.1 yices=no
```

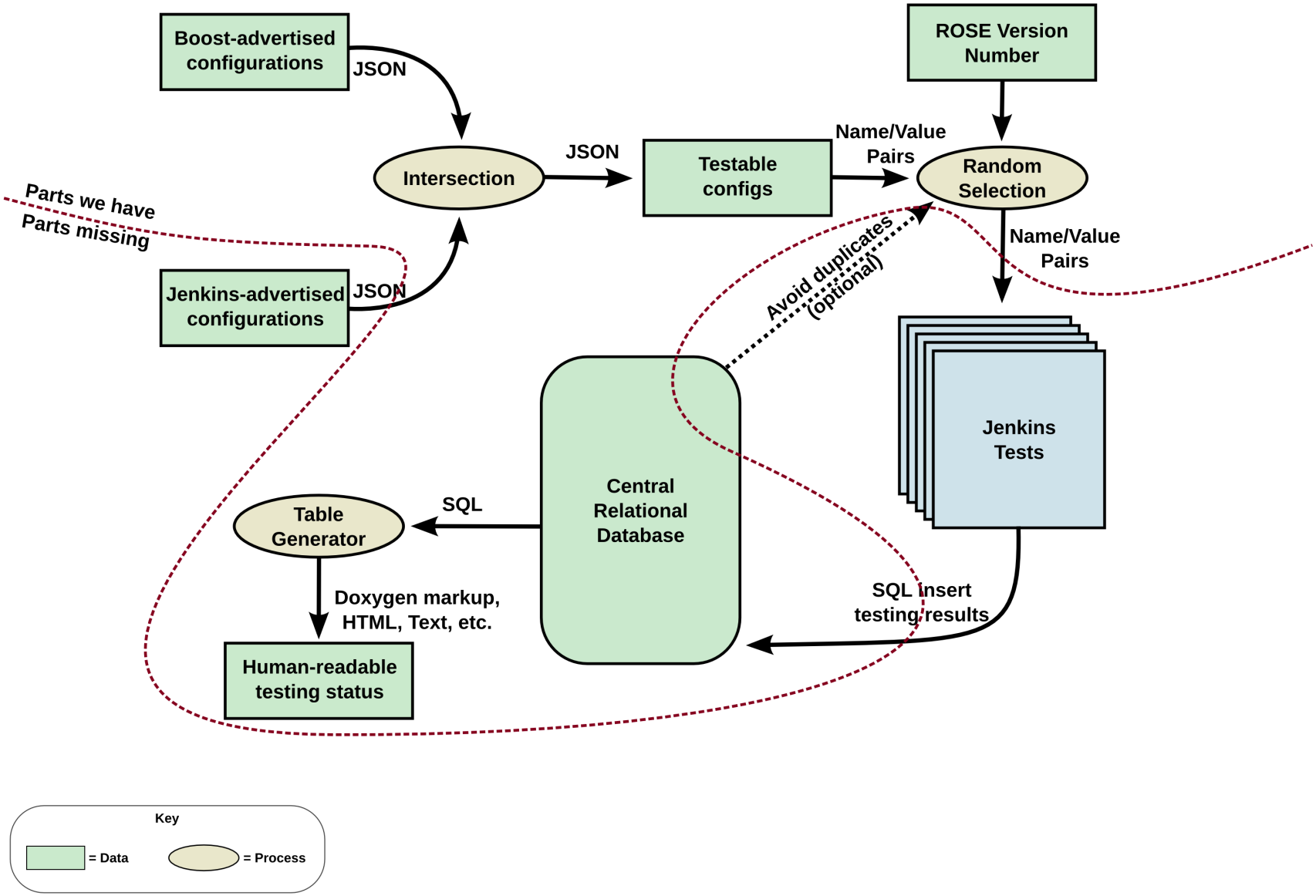
Architecture

The figure on the next page shows the basic components of the architecture. The design is intended to be:

- **Modular**: The central database is the only required component. For instance, on I run my own configuration testing on my development machine and use a couple of scripts instead of a full Jenkins setup.
- **Distributed**: The design uses a client/server paradigm where the database is the server and the other parts are clients. The clients need not run on the same host as the server.
- **Parallel**: Many clients can be working on tests or querying the database concurrently.
- **Stateless**: No client context is stored in the server.

The basic flow is that a testing client chooses a configuration vector, perhaps at random, tests it, and reports the result to the database. Another client, might connect to the database to query results and present them as a table or in a web browser.

Mechanism for testing the ROSE software dependency configuration space



Testing – deciding what to test

A testing client needs to know the configuration vector before it starts, and there are a few ways to do this. In all cases, the client should at least query the central database once in order to get the list of keys in the *key=value* configuration vector, as these keys will need to be reported back to the database (along with their selected values) when the test has completed.

Randomized testing (Monte Carlo) works well when the testing space is far to large to be filled in completely. It selects configuration vectors randomly and makes no effort to prevent running tests more than once for any particular configuration vector.

Monte Carlo 1: The testing client, after obtaining a list of keys, selects random values for those keys. The value for each key is chosen from a list of available values for the key. For instance, in the diagram above, the values for the "boost" and "compiler" keys may come from the intersection of boost/compiler combinations tested by the Boost team and the boost/compiler combinations that are installed at the testing client.

Monte Carlo 2: The testing client can query the database for a random vector. The database has a set of possible values for each key and chooses randomly from those sets. There is no requirement that the client use the vector provided by the server; the client can ask for another vector, or it can modify values for any of the keys in the vector. For instance, the database might not know about compiler patch numbers and give a configuration vector containing `compiler=gcc-4.8-c++11`, which the client can replace by filling in the patch number: `compiler=gcc-4.8.4-c++11`.

Exhaustive testing: If the space is small enough, we can test all possible configuration vectors and will want some mechanism that ensures that each vector is tested exactly one time. This is currently outside the design scope of the central database, which is stateless. Some separate mechanism needs to be written that would generate the list of configuration vectors and keep track of which tests are in progress versus completed. Testing clients would each request a vector from this service, work on it, report the result to the central database, and report completion to the vector service.

Here's an example using Monte Carlo 2: the client runs the `projects/Matrix/matrixNextTest` command to obtain a vector, which it then converts to a `configure` or `cmake` command.

```
configure_command=$(build_configure_command $(matrixNextTest -format=shell))
```

The `build_configure_command` is a shell function that creates a GNU configure command-line from a configuration vector. It is responsible for, among other things, deciding which installation of boost to use based on the compiler, deciding which installation of yaml to use based on boost and the compiler, etc. The output of `$(matrixNextTest --format=shell)` is a configuration vector presented as a list of *key=value* pairs, perhaps:

```
assertions=abort boost=1.52 build=autoconf compiler=gcc-4.8 debug=yes dlib=18.17 doxygen=1.8.9 edg=4.7
languages=binaries magic=none optimize=no python=/usr/bin/python3 qt=none readline=none sqlite=none
warnings=yes wt=3.3.4 yaml=0.5.1 yices=1.0.34
```

Testing – running the test

The mechanism used for running the test is outside the scope of this system. The testing clients could be as complex as a coordinated collection of Jenkins jobs that methodically and exhaustively test the configuration space, or as simple as a terminal window on an office machine looping over a script that uses the Monte Carlo 2 method.

In its simplest form, a test could report a simple Boolean succeeds vs. fails, but a more useful approach is to partition the test into a sequence of subtests and to eventually report the name of the subtest that failed. Subtests could be:

`setup`	Test that the configuration vector is appropriate for this testing client. For instance, does the specified compiler even exist here? This step may (silently) change the values of the *key=value* configuration vector in order to make them value (assuming the client is part of some exhaustive testing).
`configure`	Runs ROSE's configure or cmake command.
`library-build`	Runs “make -C src”
`libtest-build`	Runs “make -C tests”
`libtest-check`	Runs “make -C tests check”
`project-bintools`	Runs “make check” in various binary analysis project directories
`project-compass2`	Runs “make -C projects/compass2”
`end`	Does nothing. This is useful because the testing client will report the name of the last subtest that was started (i.e., the one that failed).

It doesn't matter what subtests are defined—the database simply stores a string for the “status” of each test. However, since one of the the database's purposes is to store a history, we should be careful that the meaning of the subtest names throughout the recorded history.

Testing – reporting results

Whenever a test completes, regardless of its status, it can be reported to the central database. There are two ways to do this:

1. Use the `matrixTestResult` tool in `projects/MatrixTesting`.
2. Send a properly formatted email to rose@hoosierfocus.com.

Either way, the result consists of the configuration vector as a list of *key=value* pairs along with some additional *key=value* pairs. The configuration vector must be the vector that was actually tested, not necessarily the one originally obtained from the database. For instance, if the original vector contained `compiler=gcc-4.8-c++11` the completion report should contain the full compiler version number: `compiler=gcc-4.8.4-c++11`.

The additional *key=value* pairs are described in the man page for the `matrixTestResult` tool obtained by running the tool with the "`--help`" switch, presented here:

MATRIXTESTRESULT(1) ROSE Command-line Tools MATRIXTESTRESULT(1)

Name

matrixTestResult - update database with test result

Synopsis

matrixTestResult [switches] key_value_pairs

Description

Adds a test result to the database. The arguments are "key=value" pairs where the keys are names of software dependencies, configuration names, or special values. The software dependency and configuration names can be obtained by querying the database "dependencies" table. The special values are:

duration

Elapsed testing time in seconds.

noutput

Number of lines of output (standard error and standard output) produced by running the test

nwarnings
Number of lines of output that contain the string "warning:".

os Name of the operating system. A reasonable value is chosen if this key is not specified on the command-line.

rose
The ROSE version number, usually a SHA1 for a Git commit object.

rose_date
The date that the ROSE version was created in seconds since the Unix epoch.

status
The final disposition of the test; i.e., where it failed. This should be a single word whose meaning is understood by the test designers and users.

tester
The entity that performed the testing, such as a Jenkins node name.

Switches

General switches

--assert how
Determines how a failed assertion behaves. The choices are "abort", "exit" with a non-zero value, or "throw" a rose::Diagnostics::FailedAssertion exception. The default behavior depends on how ROSE was configured.

--help; -h
Show this documentation.

--log config; -L config
Configures diagnostics. Use "--log=help" and "--log=list" to get started.

--threads n
Number of threads to use for algorithms that support multi-threading.

--version; -V
Shows version information for various ROSE components and then exits.

Tool-specific switches
--database uri; -d uri
URI specifying which database to use.

SQLite3
The uniform resource locator for SQLite3 databases has the format "sqlite3://filename[?param1[=value1][&additional_parameters...]]" where filename is the name of a file in the local filesystem (use a third slash if the name is an absolute name from the root of the filesystem). The file name can be followed by zero or parameters separated from the file name by a question mark and from each other by an ampersand. Each parameter has an optional setting. At this time, the only parameter that is understood is "debug", which takes no value.

PostgreSQL
The uniform resource locator for PostgreSQL databases has the format "postgresql://[user[:password]@][hostname[:port]][/database][?param1[=value1][&additional_para where user is the database user name; password is the user's password; hostname is the host name or IP address of the database server, defaulting to the localhost; port is the TCP port number at which the server listens; and database is the name of the database. The rest of the URI consists of optional parameters separated from the prior part of the URI by a question mark and separated from each other by ampersands. The only parameter that is understood at this time is "debug", which takes no value and causes each SQL statement to be emitted to standard error as it's executed.

--dry-run
Do everything but update the database. When this switch is present, the database is accessed like normal, but the final COMMIT is skipped, causing the database to roll back to its initial state.

Tests can also have attachments. Storing the last few lines of a failed test or the list of commands that were run can be useful. The `matrixAttachments` tool is used for this purpose. See the documentation from its "`--help`" switch for details.

Querying results – the command-line

The `matrixQueryTable` tool in `projects/MatrixTesting` reports results in the form of a table. It's arguments are *key=value* pairs to restrict the listing to only those tests that have all the specified pairs, and/or a list of keys to report as table columns. The columns are reported in the order specified and each is sorted subject to the previous columns. Invoking the command as "`matrixQueryTable list`" returns the set of all keys. The set of keys is a superset of the configuration vector keys and the keys reported with test results. An extra "totals" column indicates how many tests matched the criteria.

For example, to get a list of boost versions and test status you could run this command:

```
$ matrixQueryTable boost status
+-------+---------------+--------+
| boost | status        | totals |
+-------+---------------+--------+
| 1.50  | configure     | 38     |
| 1.50  | end           | 4      |
| 1.50  | libtest-build | 4      |
| 1.50  | libtest-check | 13     |
| 1.51  | configure     | 27     |
| 1.51  | end           | 8      |
| 1.51  | library-build | 1      |
| 1.51  | libtest-build | 1      |
| 1.51  | libtest-check | 14     |
| 1.52  | configure     | 23     |
| 1.52  | end           | 8      |
| 1.52  | libtest-build | 3      |
| 1.52  | libtest-check | 17     |
| 1.53  | configure     | 21     |
| 1.53  | end           | 9      |
| 1.53  | library-build | 1      |
| 1.53  | libtest-build | 2      |
```

```
| 1.53  | libtest-check | 15     |
| 1.54  | configure     | 16     |
| 1.54  | library-build | 10     |
| 1.55  | configure     | 29     |
| 1.55  | end           | 6      |
| 1.55  | libtest-build | 1      |
| 1.55  | libtest-check | 17     |
| 1.56  | configure     | 27     |
| 1.56  | end           | 3      |
| 1.56  | libtest-check | 13     |
| 1.57  | configure     | 27     |
| 1.57  | end           | 5      |
| 1.57  | libtest-build | 19     |
| 1.58  | configure     | 26     |
| 1.58  | end           | 5      |
| 1.58  | libtest-build | 14     |
| 1.59  | configure     | 17     |
| 1.59  | end           | 8      |
| 1.59  | libtest-build | 18     |
+-------+---------------+--------+
```

Or to see how boost 1.55 fared with various compilers[1]:

```
$ matrixQueryTable boost=1.55 compiler status
  boost            = "1.55"
+--------------------+---------------+--------+
| compiler           | status        | totals |
+--------------------+---------------+--------+
| gcc-4.8.4          | configure     | 2      |
| gcc-4.8.4          | end           | 3      |
| gcc-4.8.4          | libtest-check | 6      |
```

1 Part way through the history, the names of compilers were changed to include a second hyphen followed by the language: “default”, “c++11”, etc. Ideally, we should have modified the database to add “-unknown” (or “-default” if we know they all used the default language) to the compiler values that didn't have a second hyphen.

```
| gcc-4.8.4-default  | end           | 2      |
| gcc-4.8.4-default  | libtest-check | 7      |
| gcc-4.9.2          | configure     | 12     |
| gcc-4.9.2-default  | configure     | 8      |
| llvm-3.5.0         | configure     | 7      |
| llvm-3.5.0         | libtest-build | 1      |
| llvm-3.5.0-default | end           | 1      |
| llvm-3.5.0-default | libtest-check | 4      |
+--------------------+---------------+--------+
```

Test results can also store attachments. To see a list of attachments one must first know the unique test ID number by displaying the “id” key's values:

```
$ matrixQueryTable id status |tail
| 960 | libtest-build | 1      |
| 962 | configure     | 1      |
| 964 | end           | 1      |
| 966 | libtest-build | 1      |
| 968 | end           | 1      |
| 970 | libtest-check | 1      |
| 972 | libtest-check | 1      |
| 974 | end           | 1      |
| 976 | end           | 1      |
+-----+---------------+--------+
```

Then use the `matrixAttachments` tool to first list the attachments for the test, and then display a specific attachment:

```
$ matrixAttachments 972
[272]  "Final output"
$ matrixAttachments 972 272 |tail
Makefile:2342: recipe for target 'test2006_135.c.passed' failed
```

```
make[4]: *** [test2006_135.c.passed] Error 1
Makefile:2478: recipe for target 'check-local' failed
make[3]: *** [check-local] Error 2
Makefile:2195: recipe for target 'check-am' failed
make[2]: *** [check-am] Error 2
Makefile:1297: recipe for target 'check-recursive' failed
make[1]: *** [check-recursive] Error 1
Makefile:1631: recipe for target 'check-recursive' failed
make: *** [check-recursive] Error 1
```

Querying results – web browser

This is not written yet, but it would be nice to be able to present results in a web browser so users can see what configurations of ROSE are supported without first having to compile ROSE to get the `projects/MatrixTesting` tools.