l3build — A modern Lua test suite for TEX programming

Frank Mittelbach, Will Robertson and The LATEX3 team

Contents

1 Introduction

Regression tests are an important tool in any moderately complex programming environment. They allow the programmer to make extensive changes to their code while providing confidence that something that used to work still does. Extensive regression test suites have been an essential component of the maintenance and development of LATEX 2ε and LATEX3.

A regression test suite is typically composed of a number of individual files that contain one or more testable units of the code being tested. A testable unit might be either a certain computation with an expected outcome, a series of logic tests, or — in particular for TEX-based code — material that is typeset and intended to achieve some particular formatting.

During code development and before any new code is released to the public, this test suite can be compiled to ensure that any changes to the code have not introduced bugs or changed the behaviour compared to previous versions. As bugs in the code are reported, minimal examples demonstrating the bug often form test files of their own, showing that the bug has been fixed and won't re-occur.

As TEX-based code operates in at least three different 'modes' (mouth, stomach, and output), regression testing is more complex than simply asserting the outcome of certain programming logic. As part of the work of the LATEX3 project, a new Lua-based testing environment has been written to support ongoing development. This testing environment, presented at the 2014 TUG conference in Portland [3], is suitable for use by the general TEX community.

2 History

The ideas for a regression test suite for LATEX date back to the early nineties[1] when LATEX 2.09 existed in various incompatible flavours around the world due to its limitations in properly supporting font selection, complex mathematics, and languages other than English. Because of that situation LATEX 2ε was designed and implemented to reunite the different format and to provide a stable platform for future LATEX development.

However, to successfully introduce LATEX 2ε as an accepted successor of LATEX 2.09 it was essential to win over the huge LATEX user base and provide them with a system that was as stable and upward compatible as possible. Thus existing user interfaces should be preserved and typesetting should provide identical output except in those cases where bug fixes or deliberate design decisions resulted in changes.

To achieve this we devised a validation mechanism that could be used to ensure that interfaces behave as expected and typesetting results do not change even though the underlying code gets modified. With this in place the LATEX3 Project Team together with additional volunteers set out to create a large number of test files and verify them against the current LATEX 2.09 implementation. Figure 1 shows the original request for volunteers (exhibiting a severe underestimation of the amount of work involved); see also [2] for a more extensive description of this endeavor.

This effort resulted in something like 200 test files that were then used to assure ourselves that the new LATEX 2ε implementation was faithfully supporting all interfaces — it was one of the key factors that ensured the new system became an accepted replacement for LATEX 2.09 within a reasonably short period.

Once in place this regression test suite was augmented over time and now contains roughly 350 test files altogether. Whenever a bug was found and fixed we added a new test file that would exhibit the undesired behavior if that bug would somehow resurface through later changes.

Though not perfect (after all we introduced a number of bugs that initially were not caught by the

[1] As with many ideas in the TEX world, this one too can be partly traced back to Don Knuth, who already provided his own regression test for TEX a decade earlier [1].

Validating LaTeX 2.09

Writing test files for regression testing:
checking bug fixes and improvements to verify that they don't have undesirable side effects;
making sure that bug fixes really correct the problem they were intended to correct;
testing interaction with various document styles, style options, and environments.
We would like three kinds of validation files:

1. General documents.
2. Exhaustive tests of special environments/modules such as tables, displayed equations, theorems, floating figures, pictures, etc.
3. Bug files containing tests of all bugs that are supposed to be fixed (as well as those that are not fixed, with comments about their status).

A procedure for processing validation files has been devised; details will be furnished to anyone interested in this task. Estimated time required: 2 to 3 weeks, could be divided up.

Figure 1: Original request for volunteers

regression test suite), the approach served us very well and prevented a number of horrible mistakes that would otherwise have made it into public releases of LaTeX.

2.1 The needs in the '90s

With the initial regression test suite we solved a number of burning problems. First of all we wanted to be confident that the code and the documented user interfaces worked as expected. Whenever we recoded an internal function the test suite would automatically alert us if that resulted in any noticeable changes at the user level or in downright bugs.

Furthermore LaTeX 2_ε came with much more documentation and the tests included compiling and checking the documentation files for errors and missing references.

In addition the Makefiles that ran the tests also included goals to build the distribution automatically. Compared to LaTeX 2.09, which consisted of very few files, the format for LaTeX 2_ε was generated from many source `.dtx` files, so the housekeeping complexity was greatly increased.

Another issue we had to tackle was that the code was no longer maintained by a single person but by developers living in different places around the world and using different operating systems and installations. So the regression suite had to function with different installations without creating spurious differences.

Finally all tasks had to work without user intervention or manual work because only in that case will such a system be used on a regular basis and thus benefits be realized.

2.2 The general approach

Designing a test system for verifying TeX's typesetting behavior is not easy — how do you test for correctness and how do you ensure that the tests are repeatable over time and in different places?

The approach we came up with was to build test files that generate suitable data in their `.log` files. Suitable data would be, for example, the state of counters or dimensions produced with `\showthe`, data written with `\typeout`, and box content shown with `\showbox`. Some of the tracing parameters of TeX could be used to verify paragraph building or page breaking decisions, but something like `\tracingall` would be inadvisable, as that would show the internal coding and not the expected functionality.

The result of running such a test file would then be manually verified and stored away as a certified result. However, as many readers will already be aware, LaTeX's `.log` files contain a lot of irrelevant data, some of which differs from run to run and some of which differs when running on different installations. So to make this approach workable we introduced a cleanup step in which we modified the result files removing irrelevant material and normalized some of the remaining parts. Of course one has to be careful not to sanitize too far, but we found a number of things necessary or at least advisable, including

- shortening file path info to avoid differences between installations
- drop empty lines (different TeX implementations put in different numbers of these)
- drop line numbers in '`on line <num>`' to avoid differences just because extra lines got introduced in a test file.

Putting it all together we ended up with a system consisting of test files (with the extension `.lvt`), certified result files to compare against (extension `.tlg`) and a fairly complex Makefile and a number of Perl scripts used to run the different tasks. These tasks included running the test suite, producing the documentation and generating the distribution (ready to be shipped to CTAN). It also contained a number of special functions such as unpacking and locally installing the code, cleaning up the source directories,

checking individual test files, and producing a new `.tlg` file for a given test file.

2.3 The new needs (in the new century)

As mentioned above, the initial system served us well, when moving from LaTeX 2.09 to LaTeX 2ε and then throughout the '90s, which had very active LaTeX 2ε development with releases produced at half-year intervals.

In this century, development of the core of the LaTeX 2ε kernel has slowed to a minimum (releases are now only every couple of years and the changes are small) while it has intensified in other areas such as actively progressing the development of the LaTeX3 programming language `expl3`. With this new focus, newly important requirements for a regression test system became apparent.

Instead of a single distribution we now had to deal with a growing number of distributions: core LaTeX 2ε and its packages, Babel (with a different release cycle), `expl3` and possibly smaller and larger distributions of third party code that also wanted to benefit from a functional regression test system.

Windows and Mac OS X became the operating systems of choice for several developers and the Makefile approach of the original test suite did not work on Windows and only with modifications on Mac OS X.

Last but not least, a number of new TeX-based engines matured and people now wanted to use LaTeX and friends not only on pdfTeX but also on these new engines all of which provided additional capabilities. These new engines showed a number of subtle differences when adding data to the `.log` file, or due to extended capabilities showed additional data (such as extra nodes in listings). Furthermore the new engines still have bugs and a number of them showed up when we initially ran test files and compared their output with the certified `.tlg` data.

Thus testing became a multi-dimensional problem: one had to verify test results with several engines and it had to work on multiple operating systems. Furthermore new code sources posed new or different requirements for building a distribution or doing the testing and we soon found that the original approach made a number of hardwired decisions that were no longer applicable if the system was used with a distribution different from LaTeX 2ε.

For a short while we tried to accommodate the need for Windows support by using a set of `.bat` files in parallel with the Makefile approach but obviously that was doomed to failure, being impractical to maintain. Another avenue we explored was switching to a fully Perl-based approach (using Cons) but that again didn't work well with Windows and furthermore it would have been a solution not available out of the box on any TeX installation.

Eventually, we decided to apply the same principle used long ago with `docstrip.tex`: use the scripting language with some operating system capabilities that is available out of the box on all TeX installations. Back then the answer was that only TeX itself fit that bill and so TeX became the tool to build style files, etc., from `.dtx` sources. However, while TeX as such is too limited to be used for scripting a regression test system, we now had LuaTeX as an engine that offers a full-fledged Lua interpreter — and these days LuaTeX is part of all modern TeX installations.

Moving to Lua (or `texlua` to be precise) means that the test and distribution system is now not tied to either the operating system (as the script runs on Windows and Unix variants) or to third-party tools (as Lua is available as part of a modern TeX system).

– – ∗ – –

In the remaining sections of this article we describe the new system and how it can be applied to support arbitrary code within the TeX world.

3 Overview of the new system

To illustrate, a hypothetical package will be described that uses the new system: consider a package `abc` with a collection of source files in the following layout.

```
abc/
    abc.dtx
    abc.ins
    build.lua
    README
    testfiles/
              test1.lvt
              test1.tlg
              ...
              support/
                       abc-test.cls
```

What is added in addition to the normal source files is a short Lua script, normally called `build.lua`. Test files and their certified results are located in the folder '`testfiles/`' with extensions `.lvt` and `.tlg`, respectively. The files in `support/` (if any) are used when running the test files.

Upon running the test suite, a new folder '`build`' is created in which the package is unpacked, support files are copied across, and each test file is run in turn and compared to its original `.tlg` file. Directories and file names are adjustable and other setups are possible; the above structure is simply the default.

3.1 Modes of testing

The best way to perform regression tests for TEX programming is to use the `.log` file; only here can box content be tested, not just logical and programmatic constructs. Box content is essential for checking from the very highest level that code changes do not result in different typeset output.

TEX programming can be either *expandable* or not. Code that is expected to be expandable should be tested as such. This can be done by evaluating it within something like `\typeout` (in the case of LATEX). For non-expandable tests one should output their results to the `.log` once they have been evaluated. As mentioned earlier there are also a number of TEX tracing parameters and commands like `\showbox`, `\showlists`, or `\showthe` that can be used to generate relevant test data in the `.log` file.

To aid in producing a structured test suite we provide a number of commands for use in the test files. The `\TYPE` command is used to write material to the `.log` file; it works like `\typeout`, but it allows 'long' input. A variety of commands, following, then use `\TYPE` to output strings to the `.log` file.

- `\SEPARATOR` inserts a long line of `=` symbols to break up the output.
- `\TRUE`, `\FALSE`, `\YES`, `\NO` insert text strings for standardized comparison.
- `\ERROR` is not defined but is commonly used to indicate a code path that should never be reached.

To produce individual tests we offer the commands `\TEST` and `\TESTEXP`. These commands take two arguments: a title and the actual test body. `\TESTEXP` executes the body within a `\TYPE` command to test expandability but with `\TEST` you are responsible for generating test output using `\TYPE`, `\TRUE`, etc. as it is intended to be used for non-expandable tests. Both commands surround the generated output with `\SEPARATOR`s and display the title and a test number. Here is an example:

```
\begin{TEST}{stepping counters}
 {
  \setcounter{chapter}{2}
  \setcounter{section}{5}
  \setcounter{subsection}{4}
  \stepcounter{chapter}%
  \TYPE{\arabic{chapter}-%
    \arabic{section}-\arabic{subsection}}
  \SEPARATOR
  \refstepcounter{section}
  \TYPE{\arabic{chapter}-%
    \arabic{section}-\arabic{subsection}}
 }
```

This test will then produce the following output, as in standard LATEX only a counter directly "within" is reset to zero (e.g., the `subsection` counter is not touched when `chapter` is stepped):

```
========================================
TEST 8: stepping counters
========================================
3-0-4
========================================
3-1-0
========================================
```

(Assuming it's the eighth test in the file.)

4 Setting up the regression test system

Consider the case that a LATEX package consists of one or more `.dtx` files in a flat directory structure. By default, to set up a regression test suite, you would create a driver file named '`build.lua`' and sub-folder named '`testfiles/`' to contain the test files. An example driver file is shown in Section 4.2.

The test files can be called basically anything (but should be logical in some way), and by default have the extension `.lvt`. These are accompanied by a pre-saved `.tlg` file which contains the 'results' of the test file to be checked against subsequent compilation of that test. If a test file has different results for different engines it is possible to "certify" `.tlg` files for each engine; those then have extensions such as `.luatex.tlg`.

4.1 Creating and checking test output

The first time a `.lvt` test file is written, it will need to be compiled to obtain the necessary `.tlg` output for future tests. This is performed with:

`texlua build.lua save` ⟨*test name*⟩

(To produce an engine-specific `.tlg` file an additional ⟨*engine*⟩ argument can be given.) This task can be re-run as many times as necessary until the test file demonstrates the necessary behaviour being tested. At this point,

`texlua build.lua check` ⟨*test name*⟩

will then re-run the `.lvt` file and compare the result to the original `.tlg` output. If no ⟨*test name*⟩ is specified all tests in the test directory are run. Presuming no code has changed to affect the output of the tests, the console output of this task will show the name of the test files being processed followed by the line:

```
All checks passed
```

If only one test file is run the usual console output from the TEX compilation is also shown otherwise it is suppressed.

```
#!/ usr/bin/env texlua

 -- Build script for abc package

 module = "abc"

 -- variable overwrites (if needed)

 -- call standard script

 kpse.set_program_name ("kpsewhich")
 dofile (kpse.lookup ("l3build.lua"))
```

Figure 2: Driver file for a hypothetical abc package

```
\documentclass{breqn-test}
\input{regression-test}
\usepackage{breqn}
\begin{document}
\START
\AUTHOR{Will Robertson}
\begin{dmath}
a+b+c+d+e+f+g+h+i+j+k+l+m+
  n+o+p+q+r+s+t+u+v+w+x+y+z
\end{dmath}
\showoutput
\end{document}
```

Figure 3: Example test from breqn

These compilations take place in the subdirectory '`build/test`', and if a test fails, a diff file is deposited there with the information about what has changed in the output of the test file. Also deposited there are the full `.log` files for each ⟨*engine*⟩ (i.e., without modifications from the cleanup step) which can be helpful to debug complex issues.

4.2 An example driver file

For a simple setup such as shown in the overview in Section 3, the driver file (`build.lua`) is quite simple. An example of such a driver file is shown in Figure 2; it need do little more than inform the build system of the name of the package and perhaps set some flags or change some defaults if they are not adequate.

The main script is `l3build.lua`, which is automatically found in the `texmf` tree (via `kpsewhich`) and then loaded. Thus, there is no need to hardwire locations in the driver file and it will work on different installations.

4.3 The structure of test files

As mentioned previously, the method of using the `.log` file allows various types of tests to be conducted. The most simple test might load a package and execute some commands to produce a small amount of typeset output. A complete example of such a test is shown in Figure 3. Some points to note:

1. The first line, `\input{regression-test}` loads the necessary settings and commands to format the `.log` file properly for testing.
2. It is not necessary to load a special document class (most tests use `article` or `minimal`), but a package author may wish to adjust page margins, etc., without repeating the commands for each test. Such a special test class or package could then be kept in the `support/` directory.
3. The test begins proper at `\START` — everything before that point in the `.log` file will be ignored. This prevents, for example, package version numbers displayed while the preamble is processed from becoming part of the test. The `\AUTHOR` declaration is an optional way of indicating who might know how to fix the problem should the test begin failing.
4. In this example `\showoutput` generates the actual test data by generating a symbolic representation of the page content in the `.log` file.
5. A slightly modified version of `\end{document}` finishes the test document. Alternatively, one can end the test file with `\END` which avoids the final processing done by `\end{document}` and thus prevents unwanted material from becoming part of the test data. In this example `\END` cannot be used as that would stop the run immediately without producing a page — which is our goal here.

Not shown is the `\OMIT` ... `\TIMO` construction, which puts flags into the `.log` file between which no test comparisons will be made. This can be used around code that generates variable log information that is known to be irrelevant for the test. For example, statements like `\newlength` or `\newcounter` write some tracing information into the `.log` that shows the allocated register number. If the code gets revised these numbers might change and thereby unnecessarily invalidate the test result.

`\OMIT` can also be used before `\end{document}` if you need the final processing to happen, but want to ensure that nothing written at that time becomes part of the test.

An example of a more structured test from the LaTeX3 test suite is shown in Figure 4. Here, a number of different tests are contained within a single file, and a few of these are included in the example. The content of the test is not really important here (it is testing aspects of the integer module from `expl3`) but it does show a few best practices.

`\OMIT`/`\TIMO` is used to hide the register allocation numbers from `\int_new:N`. The first test then exercises integer addition and subtraction which is not expandable (therefore `\TEST` together with `\TYPE` is used) and it consists in fact of several small tests. The expected results are written as comments into the test file which is helpful in case it ever fails.

Converting integers is supposed to be expandable so `\TESTEXP` is used for the second test. The same is true for the case selection commands. Here the test output is generated by `\YES` or `\NO`.

Can you guess the test results, even if you are not familiar with the `expl3` language? They are shown in Figure 5.

4.4 Options

While the examples shown previously demonstrate the behaviour in the standard setup, the new build system provides significantly greater flexibility. This is achieved by providing a large number of variables that can be (re)set as necessary in the driver file. For example, the new system supports building complex distributions consisting of several modules in different directories with dependencies between them. You can also control if the processing should happen in a sandbox or if it is allowed to draw any support files needed for the tests (e.g., extra packages or classes) from the TDS tree. The latter is the default as this is better for most distributions. For details consult the documentation in [4].

There is one option that one may have to modify even for simple setups: `checkruns`. This controls the number of times each test file is run; to speed up processing it defaults to `1`. If, however, the codes require multiple runs to function (e.g., if you test material that is passed through the `.aux` file) you have to set this variable to `2` or higher to ensure that your tests actually work correctly.

5 Operating the system

As indicated earlier the system does a bit more than managing a set of test files, so here is a short description of the main tasks that can be executed once the setup is in place. Each task takes zero or more arguments as described below and is executed by running the driver file (default `build.lua`) through a Lua interpreter (`texlua`) and passing it the task name and any further argument as necessary, e.g.,

`texlua build.lua check` ⟨*test name*⟩

would run the check on ⟨*test name*⟩ using all engines. So here is the list of available tasks:

`check` ⟨*name*⟩ ⟨*engine*⟩ Without arguments, runs all test files found in the directory that contains

```
\documentclass{minimal}
\input{regression-test}
\RequirePackage{expl3}
\begin{document}
\START
\AUTHOR{Frank Mittelbach, LaTeX3 Project}
\ExplSyntaxOn

\OMIT
\int_new:N \l_testa_int
\int_new:N \g_testa_int
\TIMO

\TEST { adding~and~subtracting }
  {
    \int_zero:N \l_testa_int
    \int_add:Nn \l_testa_int { 5 * 7 }
    \int_add:Nn \l_testa_int { 15 }
 % we hope for a value of 50
    \TYPE { \int_use:N \l_testa_int }
    \int_sub:Nn \l_testa_int { 3 * 5 }
 % we hope for a value of 35
    \TYPE { \int_use:N \l_testa_int }
    \int_gzero:N \g_testa_int
    {
      \int_gadd:Nn \g_testa_int
                  { (2 + 13) / (2 * 3) }
      \int_gadd:Nn \g_testa_int { 3 }
 % we hope for a value of 6
      \TYPE { \int_use:N \g_testa_int }
      \int_gsub:Nn \g_testa_int { 5 * 5 }
    }
 % we hope for a value of -19
    \TYPE { \int_use:N \g_testa_int }
  }

\TESTEXP { converting~from~and~to~base }
  {
    \int_to_base:nn { 17 } { 8 }  ~
    \int_from_base:nn { 21 } { 8 }
  }
\TESTEXP{ Case~statements }
  {
    \int_case:nnn
      { -1 + 1 }
      { {    -1 } {  \NO }
        { 3 - 3 } { \YES } }
    { \NO }
  \NEWLINE
  \int_case:nnn
    { 7 - 2 }
    { { -1 + 3 } { \NO } }
    { \YES }
  }
% more tests here omitted
\END
```

Figure 4: Expandable and non-expandable tests

```
============================================
TEST 1: adding and subtracting
============================================
50
35
6
-19
============================================


============================================
TEST 2: converting from and to base
============================================
21 17
============================================


============================================
TEST 3: Case statements
============================================
YES
YES
============================================
```

Figure 5: Test results

the `.lvt` files. It reports progress by displaying each test file name currently processed (but otherwise hides any TEX output to avoid cluttering the screen) and at the end displays a summary indicating success or failure.

If ⟨*name*⟩ is specified, it will run only the tests for that `.lvt` file, and if additionally given an ⟨*engine*⟩ name will run only the test for that specific engine. In either case it will show everything on the screen, which is helpful if the run shows abnormal behaviour (especially if it ends up in an endless loop and never returns for some reason).

`clean` Cleans up the source tree, removing temporary files and directories.

`ctan` Runs all tests, typesets all documentation and if there are no errors, generates a `.zip` file suitable for uploading to CTAN.

`doc` Typesets all documentation (by default `.dtx` files), thus checking them for trivial processing errors.

`install` This installs the distribution in the local tree of the user.

`save` ⟨*name*⟩ ⟨*engine*⟩ This generates (or regenerates) the `.tlg` file for ⟨*name*⟩. If additionally supplied an ⟨*engine*⟩ argument it generates a specific `.tlg` as discussed above.

It is the responsibility of the developer to verify that the data placed into the `.tlg` produces the desired result, i.e., is actually correct. Once produced or updated with `save`, the output is considered certified and will be used to verify future check runs!

6 Acknowledgements

The original test suite system was a joint effort by the whole LATEX project team at that time, i.e., Frank Mittelbach, Rainer Schöpf, David Carlisle, Michael Downes, Alan Jeffrey, and Chris Rowley. We also had significant help when writing the initial set of test files from a number of volunteers, in particular Daniel Flipo and Chris Martin.

Around 2008 Rainer replaced the Makefile approach used for LATEX 2ε by Cons (a Perl-based solution) as the Makefile got so complex over time that it was difficult to manage.

For the LATEX3 development we stayed with Make as the requirements of the `expl3` distribution were initially much simpler.

Joseph Wright wrote a first set of `.bat` files for `expl3`, as by then many developers worked on Windows. Modelled after this, Frank replaced the Cons solution for LATEX 2ε in 2013.

Finally in 2014 Joseph then implemented most of the new Lua-based system and it is now successfully used to manage the LATEX3 (`expl3`) distribution as well as several smaller package distributions. The LATEX 2ε distribution will follow shortly.

References

[1] Donald E. Knuth. A torture test for TEX. Report STAN-CS-84-1027, 1984.

[2] Frank Mittelbach. A regression test suite for LATEX 2ε. *TUGboat*, 18(4):309–311, December 1997. `http://tug.org/TUGboat/tb18-4/tb57mitt.pdf`

[3] Frank Mittelbach. A modern regression test suite for TEX programming, July 2014. Talk given at TUG conference in Portland. Video and slide material available at `http://www.latex-project.org/papers`.

[4] LATEX3 Project. The l3build package: Checking and building packages, September 2014. `http://ctan.org/pkg/l3build`

◇ Frank Mittelbach
Mainz, Germany

◇ Will Robertson
School of Mechanical Engineering,
The University of Adelaide,
Australia

◇ The LATEX3 team
`http://www.latex-project.org`