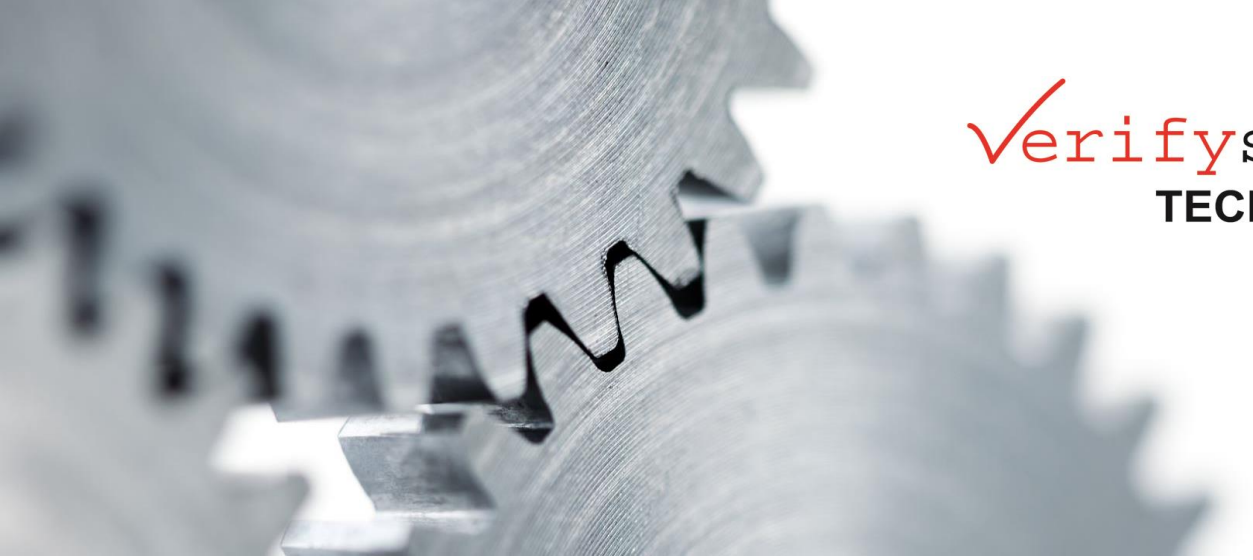

Verifysoft TECHNOLOGY

Integrated!

Testwell CTC++ with Bazel

TWEAG by Modus Create

Testwell CTC++ provides several generic ways to integrate coverage measurement in various build systems. Bazel is a famous challenge for interaction with third party tools.

Johan Herland and Mark Karpov, working for Tweag, a Modus Create company, describe their road to establish a stable integration of Testwell CTC++ with Bazel for a client of Tweag's Scalable Builds Group.

Testwell CTC++ vs Bazel's built-in coverage support

Bazel does include some support for code coverage reports, however, it is relatively limited and inflexible, only supporting GCOV/LCOV formats. Testwell CTC++, on the other hand, uses its own format, and overall uses a different approach and workflow to coverage analysis than the GNU tool suite: Testwell CTC++ runs as a *wrapper* around your usual C/C++ compiler and linker.

We quickly gave up on trying to integrate Testwell CTC++ into Bazel's existing coverage functionality for a few reasons. Instead, we focused on how to configure the build to run Testwell CTC++ in the way it wants be run: as a wrapper around the compiler/linker.

We ended up writing a custom Bazel toolchain generator for Testwell CTC++. This is a Bazel repository rule that copies the `@local_config_cc` toolchain rules but replacing its compiler/linker commands with our own wrapper script.

Our wrapper script is a small shell script that sets up the environment and options required by Testwell CTC++, and finally ends up forwarding all the compiler/linker arguments from Bazel onto the ctc command-line.

Bazel's mantra of "{ Fast, Correct } - Choose two" relies on having complete knowledge of the build graph. This includes explicitly stating the inputs and outputs of all build steps. With our new tool-

chain that invokes Testwell CTC++, we have in effect added a new output file on to every compilation step (`MON.sym`), and another output file on to every unit test step (`MON.dat`). Moreover – unless we configure Testwell CTC++ otherwise – these files are shared outputs across all compilation/test steps[1], something that we cannot easily encode into Bazel's build graph at all[2].

In the end we have not instrumented Bazel to keep track of the `MON.sym/MON.dat` files. They are not known to Bazel at all, and we instead rely on Testwell CTC++ coverage builds and test runs to always be performed from a clean source tree, with no reuse of build artifacts at all. In our case this works out well, since Testwell CTC++ coverage builds/runs are ultimately run automatically as part of the CI infrastructure where we can ensure these conditions are being met.

Navigating the Bazel sandbox

It is worth explaining some complications introduced by the sandbox in which Bazel runs its build steps. Our project builds on Linux, and we thus use the Linux-sandbox flavor of sandboxing provided by Bazel. This includes using Linux namespaces to isolate each build step, both from other build steps, as well as from the surrounding system. This is a powerful mechanism that helps achieve multiple objectives:

- Making sure all build steps are completely specified in Bazel, by cutting off access to anything that is not explicitly declared. This also helps improve the reproducibility of build artifacts, and thus the reusability/cacheability of intermediate build products.
- Preventing unwanted details from the surrounding system from leaking into the build product (e.g., hostnames, timestamps, dependencies on system libraries, etc.)
- Preventing the build from polluting the surrounding system (e.g., writing files in places where they don't belong)

However, when throwing Testwell CTC++ into the mix, there are certain concessions we have to make in order to make everything work together:

Keeping MON.sym/MON.dat outside the sandbox

In order for the writes `MON.sym/MON.dat` to be reflected outside the sandbox, we direct them to a shared directory outside the sandbox, and additionally pass this directory to Bazel's `--sandbox_writable_path` option in order for these writes to be allowed inside the restricted sandbox environment.

Preventing corruption of Testwell CTC++ temporary files across sandboxes

While Testwell CTC++ is running, it makes use of some temporary files. These are put in `/tmp` by default, and while Testwell CTC++ takes care to use the current process id (PID) when naming

[1] Don't worry, Testwell CTC++ uses **flock()** to protect against concurrent writes to these files when multiple compilations/tests are running concurrently.

[2] If we could declare these extra outputs properly to Bazel, they would surely wreak havoc on any calculations Bazel does to reuse intermediate build products for faster incremental rebuilds…

these files (to prevent concurrent Testwell CTC++ processes from using the same filenames), there is an unfortunate interplay with Bazel's sandboxing that causes problems here:

Bazel's sandbox includes a PID namespace to ensure that processes inside one sandbox cannot "see" processes in other sandboxes. However, this namespace causes the PIDs inside each sandbox (as seen from inside the sandbox) to restart their numbering from 1.

When combined with the fact that Bazel by default does not isolate the `/tmp` directory between sandboxes, this causes some temporary files to get corrupted by simultaneous writes from different Testwell CTC++ process, which then led to corrupted data being copied into the `MON.sym/MON.dat` files.

The solution we landed on here was to direct Testwell CTC++ (via the `TMP_DIRECTORY` configuration option) to write its temporary files inside the sandbox (instead of in `/tmp`) which immediately resolved the corruption.

Mapping sandbox paths to source tree paths

Another complication introduced by Bazel's sandbox is that the source file paths that are recorded inside `MON.sym/MON.dat` reference the sandbox directories that are created and deleted by Bazel before/after each build step. We originally believed this alone was enough reason to disable Bazel's sandboxing altogether (which would have caused other problems, not worth going into here), but fortunately the **ctcreport** tool comes with a `-map-source-identification` option that allows the sandbox paths to be mapped back into persistent/real source paths.

Thus Testwell CTC++ quite elegantly allows us to build the source code from sandboxes with highly variable (but predictable) names, while still allowing all source file references to be resolved back to their canonical location and be successfully found at report generation time.

All scripts used in this setup can be shared by Verifysoft on request.

About Tweag's Scalable Builds Group

We believe that correct, efficient, and reliable builds are critical for developers to work and collaborate effectively. And that the size and complexity of a project should not be bounded by its build system, but by what is best to achieve the goal of the project.

Whether you have a large codebase or a small one, whether your project is polyglot or monolingual, and whether you work in an enterprise organization or on an open source project - the build system you have available should provide correct, efficient, reliable builds.

...find out more at Tweag's Scalable Builds Group

Services from Verifysoft