Build System Schism: The Curse of Meta Build Systems

In which I talk about why meta build systems are forever cursed to remain hobbled throughout eternity.

March 19, 2024 | 11 min | 2245 words

> **Note**
>
> **Assumed Audience**: Programmers and anyone who has used a build system.
>
> Discuss on Hacker News and Reddit.
>
> **Epistemic Status**: Confident.

> **Warning**
>
> This post is meant to be informative, but it does have an ad at the end, which will be clearly marked. You have been warned!

Introduction

First there was nothing…

…then there was `make`.

As the legend goes, Stuart Feldman was working at Bell Labs, and he had a problem.

Feldman spent part of an evening struggling with an executable that was n- dated from changed source files. The next morning, Steve Johnson just happened to storm into his office to rant about the very same problem.

With his previous evening in memory, and a respected colleague wishing for something better, Feldman thought, 'You know, I could solve this with a tool.'

As it turns out, Stuart Feldman sat down and did some *design*:

> It began with an elaborate idea of a dependency analyzer, boiled down to something much simpler, and turned into Make that weekend.

We should all be grateful for that bit of design; I just wish he had gone further and not used tabs.

Or, you know, just did the work to update ***A DOZEN USERS***!

No, I'm not kidding:

> Why the tab in column 1? Yacc was new, Lex was brand new. I hadn't tried either, so I figured this would be a good excuse to learn. After getting myself snarled up with my first stab at Lex, I just did something simple with the pattern newline-tab. It worked, it stayed. And then a few weeks later I had a user population of about a dozen, most of them friends, and I didn't want to screw up my embedded base. The rest, sadly, is history.

Two lessons here:

1. Simple implementations lead to disasters, and
2. Breaking compatibility is sometimes necessary; do it as early as possible.

Nevertheless, for a while, everyone was happy!

From Make to Meta

About the time of the launch of Linux, David Mackenzie ran into a problem with Make: he needed Turing-completeness.

See, Make was designed to have a list of targets, or things to build, and that's all the targets you get, modulo suffix rules and pattern rules.

> **Note**

> Suffix rules and pattern rules were probably added just to remove repetition; programmers *hate* repetition.

But what if you didn't know what targets you needed until after some analysis?

Mackenzie had a crucial decision to make (pun intended): would he start over *without* Make, or would he build something on top of Make?

Well, I guess that something made him loathe to give up Make because he chose the latter and made Autoconf.

> **Note**
> I have no idea what it was, so I can only assume that since 1976, the legends of Bell Labs were already legends, and their software considered untouchable.
>
> It's not, by the way. And I'm generally a fan of the Unix Philosophy.

Whatever the case, this is the decision that created the **Build System Schism**.

The Build System Schism is the separation between "build systems," which includes things like Make and Ninja, and "meta build systems," which includes things like Autoconf, Automake, CMake, qmake, and Meson.

What's the difference?" you may ask.

To be overly concise, meta build systems build builds, and build systems build.

To be more understandable, meta build systems generate files that tell build systems how to execute the build, and build systems execute that build.

To be perfectly precise, let me describe the process.

Say I have a project that uses Make, but it has some build options. These build options could require anything, such as *not* compiling certain files in certain cases.

Now, let's say that I want to have my software build portably on any POSIX system, so I need to use POSIX Make.

Now I have a problem: POSIX Make is *incredibly* limited; it is not Turing-complete.

“Why is that a problem, Gavin?”

Because many times, setting up a build requires full Turing-completeness.

This post is not the place to explain why Turing-completeness matters, but it is sufficient to say that there is *no* substitute for Turing-completeness because any substitute becomes Turing-completeness by definition.

> **Note**
> Tune in next week for a deep dive into Turing-completeness, as well as what it means for build systems!

So I need something *on top of* POSIX Make, something portable.

Well, CMake is portable, so let’s say I use that.

> **Note**
> That project uses a custom shell script, not a “real” meta build system like CMake, but the example still holds.

I write a bunch of code in CMake that tells it what targets to add to the build based on the build options.

This is possible in CMake because it is Turing-complete.

> **Note**
> And if I remember correctly, CMake wasn’t even *supposed* to be Turing-complete! That’s why its files are called `CMakeLists.txt`; it was just supposed to be a list.
>
> Unfortunately, I don’t have the source, so if anyone on Hacker News or Reddit does, please tell me; I’ll add it.

So a build would look like this:

1. Run CMake, telling it the configuration to use.

2. Run Make or Ninja to do the actual build.

The Barrier

“That seems simple enough, Gavin. What’s the problem?”

The problem is that the list of items to build cannot change!

Say you add a new source file. Wouldn’t it be great if your build system just picked it up?

Alas, it cannot; the list of stuff to build is passed from the meta build system to the build system, usually by fiat.

The barrier between step 1 and step 2 is essentially unbreakable since it is the interface between two separate programs.

In fact, the only way to break it is to run the meta build system on every build.

And in that case, why not just make one build system that does both?

End-to-End Build Systems

However, there seems to be some wind shifting; there is a new type of build system coming to town.

These build systems, which I call “end-to-end build systems,” or E2E build systems, to distinguish from “regular” build systems, can not only generate the list of items to build, they can execute the build.

One example is `build2`.

> **Note**
> The creator of `build2` is borisk, from whom I shamelessly stole the idea for this post.
>
> borisk, if this post is not sufficient, feel free to tell me!

In somet like `build2`, you can specify how to search for source files and how to build ones that are found, and it will do that search on every build.

The advantage of this is that if you add another source file, *you don't have to change your build!* An E2E build system will detect the new file and build it.

In contrast, while using CMake, I either have to change one or more `CMakeLists.txt` files or run CMake manually.

Removing that barrier improves the user experience.

Building Better Build Systems

But there is a further advantage of E2E build systems: dynamic targets.

Generally, when you specify a target, you explicitly write out its dependencies in some way, either explicitly or by passing a list of dependencies in some form.

This is good, but "good" is not good enough. I want UNLIMITED POWAAAAHHH!

So what if you didn't need to specify all of the dependencies of a target? What if you could find them during the build itself using the source code itself?

At this point, *so* many people are laughing at me. "But Gavin, we already have that in `<language>`."

Yes, you do because `<language>` has a built-in build system and good module support.

As an example, if you run `go build` on a Go project, it will figure out what is imported and needs to be built, and then it will just do it. Add an `import` statement, and the next build will detect that and build the new import.

Piece of cake, right?

Well, this is still a new concept for the C and C++ world. In fact, generalizing it would probably sound foreign to a lot of languages that *do* have this.

The only E2E build systems that I know of that can do this in a general way are

Shake and Buck2.

> **Note**
>
> `build2` might, but I haven't dug deep enough to check.

Shake is a good idea ahead of its time that was also hampered by the unfortunate decision to write it in Haskell and *adopt Haskell as its build language!*

Yikes.

But never mind that; Buck2 is literally built by Neil Mitchell the same guy that built Shake. And he used the same ideas.

What ideas did Neil use?

Well, he wrote them down, but I think a better introduction is perhaps one of my Top 5 Best Papers: "Build Systems à la Carte."

tl;dr: Build systems can be classified by the following six axes:

- **Persistent build information**
  - Example: Make uses file modification times.
  - Another option is hashing files.
- **Scheduler**
  - Make is topological.
  - Shake has a suspending scheduler.
  - Bazel has a restarting scheduler.
- **Dependencies**
  - These are either static or dynamic.
- **Minimal**
  - Does the build system do the minimum possible?
- **Early Cutoff**
  - If a target is updated from a dependency, and the update didn't change any-

thing, can the build system forgo building things that depend on that target?

- **Cloud**
  - Can the build system cache stuff in the cloud and use items in that cache?

You want a **minimal** build system with a **suspending scheduler**, **dynamic dependencies**, **early cutoff**, and **cloud caching**.

> **Note**
> Persistent build information doesn't matter as long as it provides the desired features, including minimal builds.

Why do you want those things? Here's why:

Minimal: This is obvious; why would you want your build to do spurious things?

Suspending Scheduler: For this one, your options are essentially a normal scheduler (topological), restarting, or suspending.

If you have a normal scheduler, you cannot have dynamic dependencies.

And if you have Turing-completeness, a suspending scheduler is easy to turn into a restarting scheduler:

```
while (restart_needed())
{
    // Build target...
}
```

But the reverse is not possible without repeating work, which means that the build system cannot be **minimal** with a restarting scheduler.

Dynamic Dependencies: Okay, you may not be convinced about dynamic dependencies yet. But if you use Rust and Go, or any language with a module system better than *header files*, you are using dynamic dependencies.

And generalized dynamic dependencies could add such a system to C and C

> **Note**

> In fact, I suspect that dynamic dependencies are *required* for the new C++ modules.

Why are they needed? Because dependencies are specified in the source code. Essentially, you have to "build" the source to figure out what it depends on.

And since you build the source to get its dependencies, you might as well store that somewhere. And when you do, your build system has to use that in the next build.

This means the build system has to:

1. Figure out that the target exists.
2. Check for dependencies from the previous build.
3. Add those dependencies to the target *after it has been created*.

That last step is the dynamic part; the user probably didn't list the dependencies in the build files, so when the build system parsed the build files, it had no idea those dependencies existed and created the target without them.

"But it could add them while parsing the build files!"

Sure, but that is special code while dynamic dependencies are more general.

In fact, build systems *without* dynamic dependencies already do this; in the words of Neil Mitchell,

> Most build systems start with a static graph, and then, realising that can't express the real world, start hacking in an unprincipled manner. The resulting system becomes a bunch of special cases.

An example: Ninja and header files.

Hacking in an unprincipled manner is not great, and it will mean that *you*, as the user, have to do some hacking too.

In other words, dynamic dependencies are essential to keep a build system easy to use. Trust me, you want them.

Anyway, where were we? Oh, yes…

Early Cutoff: If your build system can finish faster, why not?

Cloud Caching: If your build system already built something you need, but for someone else, why not use it?

The Curse of Meta Build Systems

So there are five features of build systems that you want, and the harsh truth is that meta build systems ***cannot*** have some of them! Some of these things are *only* available on end-to-end build systems.

This is the curse of meta build systems; they will forever be hobbled and hampered from reaching the full potential of build systems.

So next time you are looking for a build system, make sure you check if it is a meta build system. If it is, keep looking; there are better options.

Turing-Completeness

But there is one axis that even “Build Systems à la Carte” missed: Turing-completeness.

Unfortunately, this is one axis that most of the best build systems still get wrong. Even *Buck2* decided to go with a non-Turing-complete language.

But Turing-completeness and its consequences are so misunderstood that it deserves its own post.

Tune in next week!

> **Warning**
> Below is the ad mentioned at the top!

Rig

I have spent more than three years learning about build systems and building one.

It's called Rig, and it will be publicly released on April 2, 2024.

It will be **minimal** with a **suspending scheduler**, **dynamic dependencies**, **early cut-off**, and **cloud caching**. Obviously.

Look for an announcement on Hacker News, Reddit, and this blog!

Tech Design Programming Build Systems Make CMake Autoconf Rig