Forward Build Systems, Formally

Sarah Spall
Indiana University
USA
sjspall@iu.edu

Neil Mitchell
Meta
UK
ndmitchell@gmail.com

Sam Tobin-Hochstadt
Indiana University
USA
samth@indiana.edu

Abstract

Build systems are a fundamental part of software construction, but their correctness has received comparatively little attention, relative to more prominent parts of the toolchain. In this paper, we address the correctness of *forward build systems*, which automatically determine the dependency structure of the build, rather than having it specified by the programmer.

We first define what it means for a forward build system to be correct—it must behave identically to simply executing the programmer-specified commands in order. Of course, realistic build systems avoid repeated work, stop early when possible, and run commands in parallel, and we prove that these optimizations, as embodied in the recent forward build system Rattle, preserve our definition of correctness. Along the way, we show that other forward build systems, such as Fabricate and Memoize, are also correct.

We carry out all of our work in Agda, and describe in detail the assumptions underlying both Rattle itself and our modeling of it.

CCS Concepts: • **Software and its engineering → Formal software verification**.

Keywords: agda, build systems, concurrency, functional programming, program verification, systems, verified applications

ACM Reference Format:
Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2022. Forward Build Systems, Formally. In *Proceedings of the 11th ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP '22), January 17–18, 2022, Philadelphia, PA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3497775.3503687

1 Introduction

Build systems are used by everyone. They provide the powerful ability to describe how complex projects should be built and to build them in a repeatable and efficient fashion. Two of a build systems most important features are *incrementality* and *parallelism*. To provide both *incrementality* and *parallelism* build systems like Make [4] require the user to declare the targets to build, what those targets depend on, and how to build those targets. Assuming the user specified all the dependencies of the project correctly, the software project will be built correctly; during re-builds only those targets whose dependencies have changed will be built again; and targets which do not depend on one another can be built in parallel. But, getting the dependencies of a software project correct is not so easy, as Licker and Rice [7] show. Omitted or incorrect dependencies can lead to consequences ranging from missed opportunities for parallelism to failure to rebuild in the presence of changes (often requiring a "clean" step) to outright incorrect results.

An alternative to a build system such as Make is a *forward build system*, where a user writes a program that says how to build their software project, without declaring targets or dependencies. A *forward build system* is, conceptually, just a simple command interpreter; it takes a sequence of commands and executes them, without checking for things such as targets or dependencies, as if the build was a Shell script. Unlike a Shell however, it can provide *incrementality* and even *parallelism*. Let's take an example:

```
gcc -c file.c
gcc -c string.c
gcc -c print.c
gcc file.c string.c print.c -o program
```

A Shell script would run each of the commands in this script every time the script was run. A *forward build system* however, by using *system tracing*, runs each command and records the files the command read or wrote during its execution. When the build is re-run, perhaps after a change to one of the files, it can use this information to decide if it should run a command again, much how Make decides if it should re-build a target by checking if its dependencies have changed. Forward build system implementations include Memoize [9], Fabricate [5] and Rattle [14].

Like our notional Shell script, forward build systems are typically embedded in full-fledged languages to provide control structures, libraries, and other conveniences: Memoize and Fabricate build scripts are Python programs, and Rattle build scripts are Haskell programs. In this paper we model *forward build systems* in general, and then continue

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).
CPP '22, January 17–18, 2022, Philadelphia, PA, USA
© 2022 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9182-5/22/01.
https://doi.org/10.1145/3497775.3503687

with Rattle in more detail. Our reason for focusing on Rattle is that it contains two unique features—first, it provides support for implicit parallelism via *speculation*, and second it defines a notion of whether the commands in a forward build system are valid or conflict with each other via *hazards*.

Hazards allow stating correctness for Rattle in a way that is not possible for prior forward build systems. In particular, a build process that writes to some of the *inputs* to the build cannot be correctly executed based solely on remembering past commands, since there's no one answer as to what the past state was. In this situation as well as others, Rattle detects the hazard and reports an error to the user.

We thus aim for the following notion of correctness:

> *A forward build system is correct if, for every build, it either produces the same result as running the commands in order, or reports an error.*

Our central contribution is a simple yet formal account of what it means for a forward build system to be correct, based on the above idea. We demonstrate the utility of our approach with application to models of both simple and sophisticated forward build systems, including hazard detection, memoization, and parallel speculation.

This paper informally introduces Rattle, the optimisations is provides, and what correctness means in §2. We then move to Agda, providing:

- The key concepts and definitions underpinning all of our models of forward build systems in §3.
- A formal model of Fabricate, as a representative of simple forward build systems, and proof of its correctness in §4.
- A formal model of Rattle and proof of its correctness in §5. We prove the correctness of Rattle first without speculation, and then including speculation.

In the process of formalising these proofs we found a small bug in Rattle hazard computations, which we fixed in the proof, see §6.4. The same fix can be applied to Rattle itself, showing the value of proving these complex concepts formally.

2 Rattle

Rattle is a *forward build system* that is implemented in Haskell and whose build scripts are Haskell programs which use the Rattle API, given in Figure 1. A Rattle build script comprises of control logic in Haskell and calls to `cmd` which execute external processes. Taking our example from §1, we can write the following Rattle program, which uses both the Rattle API and the Haskell library `System.FilePath`.

```
-- The Run monad
data Run a = ...
rattle :: Run a -> IO a

-- Running commands
cmd :: CmdArguments args => args
data CmdOption = Cwd FilePath | ...

class CmdArguments args
-- instances to allow any number
-- of String/[String]/CmdOption values
```

Figure 1. Part of the Rattle API.

```
main = rattle $ do
   let cs = ["file.c" , "string.c"
            , "print.c"]
   forM cs (\c -> cmd "gcc -c" c)
   let to0 x = takeBaseName x <.> "o"
   cmd "gcc -o program" (map to0 cs)
```

This program is made up of regular Haskell logic, such as `forM` and `let`, which is not visible to Rattle, and the calls invoked by `cmd`, which are visible to Rattle. The control logic is executed every time the script is run, so Rattle's view of the script is:

```
gcc -c file.c
gcc -c string.c
gcc -c print.c
gcc -o program file.o string.o print.o
```

Whenever Rattle executes a command, it uses *tracing* to record which files were read (inputs) and written (outputs) by the command, and the contents of those files. In future executions, if Rattle sees a command which previously ran with the current value of the inputs and outputs, it skips execution, assuming that the command would have no effect. If Rattle sees a command which previously ran with the current inputs, but where Rattle has access to a copy of the previous outputs, it copies those outputs to where the command would put them, and doesn't run the command. If the command has never been seen before with these inputs, it will be run afresh with tracing.

One weakness of this approach is that the execution is single-threaded, only running one command at a time. In our example, it should be possible to run all the `gcc -c` commands in parallel, since they work on disjoint source files and destinations. To overcome that weakness, Rattle uses *speculation*, where it predicts commands that are likely to be required in future but also unlikely to conflict with other commands, and runs them *before* the script requests them. The commands Rattle considers for speculation are simply those that executed in the previous run. Of more interest is the way Rattle decides whether a command is likely to

conflict, which it does using a concept called hazards. Rattle chooses a command to speculate next by picking a command which would not cause a hazard with any command already completed, or a read conflict with any command currently running, according to the tracing data from the commands' previous run.

2.1 Hazards

A build system has reached a fixed point if on a subsequent rebuild, no work is done, because every command is already up to date. However, that's not true for all shell scripts – consider the script:

```
gcc -c foo.c
echo X >> foo.c
```

On the first execution the file `foo.c` is compiled and then, after it has been used as the input to `gcc`, is modified. The Rattle paper [14] introduced *hazards*, which detect bad behaviour, where the absence of any hazards implies the build has reached a fixed point. In particular, writing to a file after it has already been read from, as in the `echo X` command above, is a read-before-write hazard. The three kinds of hazards defined by Rattle are:

Read-before-write hazard One command reads a file which a later command writes to. On a subsequent rebuild the first command will need to run again because the second command wrote to its dependency.

Write-before-write hazard A command wrote to a file a later command writes to the same file again. On a subsequent rebuild the first command will need to run again because the second command changed its output file, and that first command rerunning will likely trigger the second command to run again.[1]

Speculative write-before-read hazard When Rattle runs commands requested by the build, it marks them as *required*. If a command was run speculatively and hasn't been marked as *required* yet, it is *speculated*. If a *speculated* command writes to a file later read by a *required* command then Rattle has run commands in the wrong order, and a speculative write before read hazard has occurred.

2.2 Assumptions

In order for a hazard-free build to have reached a fixed point, Rattle makes certain assumptions which it does not check:

Determinism of commands Rattle assumes all commands are deterministic, although it doesn't enforce this property. When a command is not deterministic it assumes all possible outputs are equivalent. Rattle supports using *hash forwarding* for non-deterministic commands, hashing the inputs of a command rather than the outputs. If a build contains non-deterministic commands then it might have a different result each time it runs, which is not the desired behavior of a build system. Note that this is assumed by *all* practical build systems, including Make—the alternative is to abandon the value of build systems entirely.

Disjoint reads and writes Rattle assumes commands do not write to their own inputs (aka all writes first truncate). If a command writes to its own inputs, Rattle cannot record the true value of its inputs since the tracing used only captures their value *after* the command has completed.

Tracing data is correct Rattle assumes tracing data is complete and correct, if it is not, then Rattle might make the wrong decision about when to re-run commands and which commands it is safe to run in parallel. Rattle supports a number of tracing backends per platform (using `LD_LIBRARY_PRELOAD`, hooking system calls, etc), which vary in their precision vs performance trade-offs, but we assume an ideal model where they are correct. For example, Rattle does not track directory operations or operations that are not reads or writes; our model does not contain directories or any such file operations. A detailed discussion of Rattle's approach to tracing is given by Spall et al. [14, §3.5]

2.3 Correctness

The Rattle implementation executes a build with speculation, and if that raises a hazard, repeats the build without speculation. Some build scripts are considered flawed, and *always* raise a hazard, even without speculation. We consider the Rattle approach to be correct if every execution *either* raises a hazard *or* produces a result equivalent to that of the shell script.

In the following sections we model Rattle in Agda, then prove that this model, including its approach to speculation, meets this definition of correctness.

3 Modeling Forward Build Systems

In this section we describe the framework in which we model both the forward build systems Fabricate (§4) and Rattle (§5). In particular we aim to prove that the build systems are equivalent to running the underlying shell script, which we define as the function shell. All these definitions are in Agda [12] [2]. The most important types are given in Figure 2.

[1] Note that file moves as commands are likely to cause hazards, under these definitions, unless the moved file was in the input, which would make the build fundamentally non-idempotent. Moves can be combined with the command that produced the moved file, however, to produce a command that Rattle accepts.

[2] The full source is available at https://github.com/spall/rattle-model/tree/paper_version_final

```
FileName : Set
FileName = String
FileContent : Set
FileContent = String
File : Set
File = FileName × FileContent
FileSystem : Set
FileSystem = FileName ⟶ Maybe FileContent
MaybeFile : Set
MaybeFile = FileName × Maybe FileContent
Memory : Set
Memory = List ( Cmd × List MaybeFile )

Cmd : Set
Cmd = String
Build : Set
Build = List Cmd
```

Figure 2. Table of key types used in our model.

3.1 Modeling Files

For any build system, files are an inherently important aspect. Therefore we define:

1. FileName, being a String which represents a path to a specific file on the file system.
2. FileContent, being a String which represents the contents of a specific file on the file system.
3. File, being a pair of FileName and FileContent which describes a particular file.
4. FileSystem, being a mapping from a FileName to a Maybe FileContent, where Nothing represents that the file does not exist in the FileSystem.

3.2 Modeling Commands

From the view of Rattle, a build is just a series of commands it is given to execute, so we model a Build as a list of commands. We define a command as Cmd, represented as a String, although the choice of representation is not too important – they need equality but little else. The actions of a command, such as `gcc -c file.c` depend on the FileSystem it is run on. It isn't sufficient for a command to be modeled by something static, the result of the command depends on `gcc` and `file.c`, and in particular the header files accessed depend on the contents of `file.c`. We model the effect of a Cmd as a function, CmdFunction, which describes the actions of the command and reports the files read and written to by the command when run on a FileSystem.

```
CmdFunction : Set
CmdFunction = FileSystem ⟶ List File × List File
```

There is an Oracle for converting a Cmd to a CmdFunction. CmdFunctions are deterministic, so the result of a CmdFunction will be equivalent for equivalent FileSystems. But, our intuition tells us a command will have the same result when run on many different FileSystem values, as long as the files the command depends on are the same. So, the Oracle maps a Cmd to a dependent product of a CmdFunction and a CmdProof. CmdProof provides us with evidence that for any two FileSystems, s_1 and s_2, the CmdFunction f will produce an equivalent result when run on both s_1 and s_2 if the files read by the command according to f have the same value in both s_1 and s_2. Although Rattle allows adjusting equality comparison using hash forwarding, thus supporting commands that are only weakly deterministic, our model assumes strong determinism (that is, that commands produce identical outputs on identical inputs) for command outputs because weak determinism would provide us with no additional expressiveness. Modeling weak determinism would require adding a function which maps all equivalent FileContents to the same FileContent, but no other changes to the model.

```
Oracle : Set
Oracle = Cmd -> Σ[ f ∈ CmdFunction ](CmdProof f)

-- names of files read according to cmdFunction
reads : CmdFunction ⟶ FileSystem ⟶ List FileName
reads f s = map proj₁ (proj₁ (f s))

CmdProof : CmdFunction ⟶ Set
CmdProof f = ∀ s₁ s₂
  ⟶ (∀ g₁ ⟶ g₁ ∈ reads f s₁ ⟶ s₁ g₁ ≡ s₂ g₁)
  ⟶ f s₁ ≡ f s₂
```

Stated alternatively, while the CmdFunction takes the entire FileSystem, its result is only dependent on the subset of the FileSystem it claims to use. This property matches the assumptions of determinism and accurate tracing from §2.2.

Running a Cmd, using run, extends the FileSystem with the files the CmdFunction writes to.

```
-- writes according to Cmd's CmdFunction
writes : Cmd ⟶ FileSystem ⟶ List File
writes = proj₂ ∘₂ (proj₁ ∘ oracle)

run : Cmd ⟶ FileSystem ⟶ FileSystem
run x s = foldr extend s (writes x s)
```

Our reference behavior is that which happens when a build is executed as if it was a script, i.e. when the commands are executed with no incrementality or other optimizations. To express that concept, we define a function script, which executes a Build by calling run on each command in the build. This script function is used to prove that optimisations preserve the reference behavior.

```
script : Build ⟶ FileSystem ⟶ FileSystem
script [] sys = sys
script (x : b) sys = script b (run x sys)
```

3.3 Preconditions and Simplifications

Our model has certain preconditions and simplifications based on Rattle's behavior, which we outline here.

Builds cannot contain duplicate commands Some of our lemmas explicitly require that a Build contain no duplicate commands. This assumption is sufficient to model Rattle builds because Rattle does not run duplicate commands. If a command occurs more than once in a build it will be skipped on subsequent appearances.

Builds are static lists Our model makes the simplifying assumption that builds are static lists. Rattle provides support for monadic builds, meaning the result of previous commands can influence the future commands run. While convenient for users, monadic commands complicate the proof, and if the proof is considered as a consequence of the initial FileSystem, provides no additional expressive power.

Commands are run sequentially Rattle can support parallel builds, but our model does not explicitly model parallelism. Instead, builds are a list of commands that run sequentially, each modifying a FileSystem in sequence. In reality, Rattle is only able to detect which files were accessed after a command completes, so to be conservative and report all possible hazards, it assumes all reads happened at the beginning of the command and all writes at the end. We simplify our model to ignore parallelism, but any successful parallel interleaving can be encoded by having multiple distinct commands writing to temporary files, which composed together form the full command (capturing the early read and late write that Rattle assumes). If the commands are not atomic, and thus have a different result when run sequentially, then there must be a *hazard* in the overall build, which would be detected in the sequentialization as well as in the actual parallel execution.

In our Agda model some of these preconditions are captured as PreCond *s br bs*, where *s* is a FileSystem, *bs* represents the Build provided as the script, and *br* represents the Build that was actually run (which may be different from *bs* if speculation was involved).

3.4 Hazards

In §2.3 we introduced Rattle's notion of correctness. We say a build system is correct if executing the build script gives identical results to the shell script and is also idempotent, or raises a hazard. We also showed there are builds which won't be idempotent because they contain sequences of commands, which modify each others dependencies in a way that can cause those commands to re-run on an immediate subsequent rebuild. Rattle considers such sequences of commands to be *hazardous*, and defines two hazards to describe these problematic sequences, *read before write hazards* and *write before write hazards*.

Hazards also enable Rattle to detect when it has made an error with speculation. When Rattle speculates commands it does so assuming the command is part of the build script and that its dependencies have not changed since it was last run. But, a speculated command might no longer be part of the build, or its dependencies might have changed, potentially leading to a *speculative write before read hazard*, where a *speculated* command wrote to a file a later non-speculated command read from, indicating Rattle did not run the commands in the order intended by the build author. For example:

```
gcc -c file.c
gcc -c string.c
gcc -c print.c
gcc file.c string.c print.c -o program
```

Let's say Rattle ran the above list of commands, speculating *gcc -c file.c*, which is no longer part of the user's build script. Then, *gcc file.c string.c print.c -o program* would read a version of *file.o* unintended by the build script's author. Through speculation Rattle inadvertently ran a command it was not meant to run, causing a later command to potentially read the wrong data.

In this section we present two data types, Hazard and HazardFree, which provide evidence when one of the three hazards has occurred, or evidence that a build contains no hazards, respectively.

The Hazard data type (Figure 3), represents a hazard occuring in the build. Hazard is indexed on a FileSystem, Cmd, Build, and FileInfo, and provides evidence of a hazard occurring after the Cmd has run on the FileSystem. The Build Hazard is indexed on, is the *script* build, the one meant to be executed, and is specified for the purpose of deciding if a *speculative write before read hazard* has occurred. FileInfo is a list, used to record the Cmds run so far, and the files they read and wrote to, for the purpose of detecting hazards.

```
-- FileNames == List FileName
FileInfo : Set
FileInfo = List (Cmd × FileNames × FileNames)
```

Hazard has a constructor for each of the three types of hazard. ReadWrite and WriteWrite construct evidence of *read before write* or *write before write* hazards respectively. A *read before write* or *write before write* hazard has occurred if the command writes to a file a previous command read or wrote to. ReadWrite constructs a Hazard by showing the writes of the Cmd intersect with the files read by the previous Cmds run, as recorded in the FileInfo. WriteWrite constructs a Hazard by showing the writes of the Cmd intersect with the files written to by previous Cmds, as recorded in the FileInfo.

```
-- The FileNames the Cmd read according to the FileInfo
cmdRead : FileInfo → Cmd → List FileName

-- The FileNames the Cmd wrote according to the FileInfo
cmdWrote : FileInfo → Cmd → List FileName

  -- extends FileInfo with a new entry for the Cmd
  save : FileSystem → Cmd → FileInfo → FileInfo
  save s x fi = (x , (cmdReadNames x s) , (cmdWriteNames x s)) : fi

data Hazard : FileSystem → Cmd → Build → FileInfo → Set where
  ReadWrite  : ∀ {s} {x} {b} {ls} {v} → v ∈ (cmdWriteNames x s) → v ∈ (filesRead ls) → Hazard s x b ls
  WriteWrite : ∀ {s} {x} {b} {ls} {v} → v ∈ (cmdWriteNames x s) → v ∈ (filesWrote ls) → Hazard s x b ls
  Speculative : ∀ {s} {x} {b} {ls} {v} x1 x2 → x2 before x1 ∈ (x : (cmdsRun ls)) → x2 ∈ b → ¬ x1 before x2 ∈ b
                → v ∈ cmdRead (save s x ls) x2 → v ∈ cmdWrote (save s x ls) x1 → Hazard s x b ls

data HazardFree : FileSystem → Build → Build → FileInfo → Set where
  [] : ∀ {s} {b} {ls} → HazardFree s [] b ls
  _:_ : ∀ {s} {x} {b1} {b2} {ls} → ¬ Hazard s x b2 ls → HazardFree (run x s) b1 b2 (save s x ls) → HazardFree s (x : b1) b2 ls
```

Figure 3. Data structures capturing hazards and the absence of hazards and associated helper functions.

Constructing evidence of a *speculative write before read* hazard is more complicated, because whether or not a *speculative write before read* hazard has occurred depends on the order the commands in the build were *meant* to run. Speculative constructs evidence of a *speculative write before read* hazard by showing there are two commands, x_1 and x_2, from those run so far (i.e. recorded in the FileInfo, or the Cmd just run, x) where the later command, x_2 read a file, which the earlier command x_1 wrote to, but x_1 was not meant to run before x_2 , possibly because x_1 was never meant to run (x before y ∈ ls says x is before y in the list ls). Unlike ReadWrite and WriteWrite, which say there exists a Hazard explicitly involving Cmd Speculative says there was a speculative hazard somewhere in the history of the build. When coming up with a representation for *speculative write before read* hazards we realized currently Rattle does not correctly detect all *speculative write before read* hazards. Our model assumes a fixed version of Rattle which correctly detects all speculative hazards. We discuss the specifics of how Rattle was wrong in §6.4.

The converse of the Hazard data type is the HazardFree data type (Figure 3), which provides evidence a build contains no hazards. HazardFree is indexed on a FileSystem, two Builds, and a FileInfo. The FileSystem is the one we are running the first Build in, the second Build is the *script* build, required to prove there are no *speculative write before read* hazards, and the FileInfo is the record of the commands run so far. HazardFree is an inductive data type with an empty constructor , [], which trivially says that an empty Build is HazardFree, and a constructor, _ :: _ which says the first command in the build is hazard free, ¬ Hazard, and the rest of the build is HazardFree after running the first command.

4 Correctness of Fabricate

A forward build system is fundamentally just a script with support for incrementality. Fabricate traces the commands it runs, and records the files they read to decide whether or not they should be run on re-builds. In this section we present a model of the forward build system Fabricate [5], as well as a *correctness* lemma and proof in Agda. Given the similarity to Memoize [9], the same model and proofs apply identically.

4.1 Modeling Fabricate

We model Fabricate by extending script, described in §3 with the use of a Memory to support incrementality. Memory is a list of Cmds and a list of File values (see Figure 2), which records the commands run and which files they read. Before we state the definition of fabricate we define a new function runF for running commands. runF checks if a command should be run, using run?, which checks if the Cmd is in the Memory and the Files recorded have unchanged values in the FileSystem and thus would have no effect if run(assuming determinism, see §2.2). If run? says the Cmd should be run, either because there is no entry for it in the Memory or the values of the FileNames stored have changed in the current FileSystem (get retrieves the files stored in the Memory for x and maybeAll checks if they have changed in the FileSystem), runF calls doRun. FunctiondoRun calls run defined in §3 and extends the Memory

```
correct-fabricate : ∀ {s} b → PreCond s b b → HazardFree s b b [] → (∀ f₁ → proj₁ (fabricate b (s , [])) f₁ ≡ script b s f₁)
```

Figure 4. A correctness lemma for Fabricate.

with a new entry for the Cmd run and the files it read; only the files read are recorded just as Fabricate does.

```
run? : Cmd -> State -> Bool
run? x (s , m) with x ∈? map proj₁ m
... | no x∉ = Bool.true
... | yes x∈ = is-nothing (maybeAll {s} (get x m x∈))

-- store extends the Memory with a new entry
doRun : State -> Cmd -> State
doRun (s , m) x = let s₂ = St.run x s in
                  (s₂ , store x (cmdReadNames x s) s₂ m)

runF : Cmd → (FileSystem × Memory)
     → (FileSystem × Memory)
runF cmd st = if (run? cmd st)
              then doRun st cmd
              else st
```

Finally, below we define fabricate. It takes a Build, FileSystem, and Memory and returns a FileSystem and Memory, using runF to run commands.

```
fabricate : Build → (FileSystem × Memory)
          → (FileSystem × Memory)
fabricate [] st = st
fabricate (x : b) st = fabricate b (runF x st)
```

4.2 Proving Fabricate Correct

In §2.3, we informally stated that a forward build system is correct if for all hazard free builds, executing a build with the forward build system has the same effect as executing it as a script. So, we state the following correctness theorem for Fabricate:

> *Fabricate is correct if for all hazard free builds, executing a build with Fabricate has the same effect as running it as a script.*

Figure 4 shows the corresponding lemma in Agda, correct-fabricate. It says for a FileSystem, s, and a Build, b, for which we know our pre-conditions are true, and which is HazardFree, the FileSystem produced by executing b with fabricate is equivalent to the FileSystem produced by executing b with script. Two FileSystems are equivalent if for all FileNames, the values are the same in both FileSystems. Essentially, fabricate and its usage of a Memory preserves the behavior of script for HazardFree builds.

We omit the details of the proof here, but it proves that for each Cmd, c, in the Build, b, the FileSystem produced by running c with runF is equivalent to the FileSystem produced by running c with run, because if c has been run before then running it is equivalent to not running it because its writes could not have changed in the FileSystem after c was last run, otherwise there would be a WriteWrite Hazard.

Of note, the proof does not require the ReadWrite hazard free property, although we would require such a property if we were to prove idempotence. The proof also doesn't require Speculative either as Fabricate does not perform speculation.

5 Correctness of Sequential Rattle

In this section we extend our model from §3 to Rattle, and state and prove a correctness lemma for sequential Rattle.

5.1 Modeling Rattle

We model two variants of Rattle in order to express the necessary proofs – rattle-unchecked which doesn't check for hazards, and rattle which does. Like Fabricate, Rattle offers incrementality through the use of memory, but unlike Fabricate, Rattle stores both the files a command read and wrote, rather than just those read.

We begin by defining a new function for *running* commands, runR, which is identical to runF, except it calls doRunR rather than doRun. We omit the body of doRunR here, because it is the same as doRun except it records the Cmd's writes in the Memory in addition to the reads. We then use runR to define rattle-unchecked almost identically to fabricate; it uses runR rather than run.

```
runR : Cmd → (FileSystem × Memory)
     → (FileSystem × Memory)
runR cmd st = if (run? cmd st)
              then doRunR st cmd
              else st

rattle-unchecked : Build → (FileSystem × Memory)
                 → (FileSystem × Memory)
rattle-unchecked [] st = st
rattle-unchecked (x : b) st = rattle-unchecked b (runR x st)
```

When Rattle executes a build, after each command finishes it checks for the hazards described in §2.1. Rattle keeps a record of the files accessed so far in the build, which command accessed the file as well as a timestamp of when the file was accessed, for a read the command's starting timestamp is recorded, and for a write the command's finishing timestamp is recorded. Rattle also keeps a record of which commands have been *required* by the build so far, meaning the build script has requested them to run, they

were not just run via speculation. To check for hazards Rattle compares the files the command which just completed accessed to the files accessed so far. If the current command wrote to a file after another command read or wrote to that file, then a *read write* hazard or a *write write* hazard has been detected, respectively. To detect if a *speculative write before read* hazard has occurred, Rattle checks if a previous command, which was not meant to run before the current command, wrote to a file it read. Rattle uses the list of *required* commands to learn the order commands were meant to run in. If both commands are in the *required* list, and the current command is first, then the commands ran in the wrong order and a *speculative write before read* hazard has occurred. If only the current command is in the *required* list, then a *speculative write before read* hazard has occurred. Commands are added to the required list when they are *required* by the build script, regardless of whether or not they were run speculatively.

Because rattle-unchecked does not check for hazards we also define rattle, which checks for the hazards described in §2.1, to more closely model Rattle. To facilitate this we define a new function for *running* commands, runWError. Just as runR, runWError uses run?, but now it also now performs hazard checking using checkHazard, before calling doRunR and adding a new entry to the FileInfo with rec. We omit the implementation of checkHazard here, but it looks for ReadWrite and WriteWrite hazards by checking if the Cmd just *run* wrote to any files recorded in the FileInfo. It checks for Speculative hazards by seeing if two Cmds exist, including the Cmd just run, where the first Cmd run wrote to a file the later Cmd read, but the first Cmd was not meant to run before the later Cmd. Unlike Rattle, the model has perfect information about which commands were meant to run and the order they were meant to run in. So, the model could detect all *speculative write before read* hazards as soon as they happen, but to more closely simulate Rattle, the model waits until the latter command could be detectably *required* by Rattle to report a *Speculative* hazard. The model checks if a Cmd has been *required* by seeing if all of the Cmds meant to run before it, are recorded in the FileInfo.

```
checkHazard : ∀ s x {b} ls → Maybe (Hazard s x b ls)
```

```
runWError : ∀ {b} x s m ls
  → Hazard s x b ls ⊎ (FileSystem × Memory) × FileInfo
runWError x s m ls with (run? x (s , m))
... | false = inj₂ ((s , m) , ls)
... | true with checkHazard s x ls
... | just hz = inj₁ hz
... | nothing = inj₂ (doRunR (s , m) x , rec s x ls)
```

Finally, we define rattle, which returns either a Hazard (proving that a hazard was reached), or a FileSystem and Memory. ∃Hazard is an abbreviation for existentials and a Hazard, because Hazard is indexed on things we cannot include in rattle's type signature.. To support speculation, rattle takes *two* builds, the build to run, *br*, and the script build supplied by the user, *bs*. In the case of sequential Rattle, these two are the same.

```
∃Hazard : Build → Set
∃Hazard b = ∃[ sys ](∃[ x ](∃[ ls ](Hazard sys x b ls)))
```

```
rattle : (br bs : Build) → (FileSystem × Memory) × FileInfo
      → ∃Hazard bs ⊎ (FileSystem × Memory) × FileInfo
rattle [] bs st = inj₂ st
rattle (x : b₁) bs st@((s , m) , ls) with runWError x s m ls
... | inj₁ hz = inj₁ (proj₁ (proj₁ st) , x , proj₂ st , hz)
... | inj₂ (st₁ , ls₁) = rattle b₁ bs (st₁ , ls₁)
```

5.2 Correctness of rattle-unchecked

Before we prove rattle is correct, we state and prove rattle-unchecked is equivalent to script; that Rattle which does not check for hazards is equivalent to the Script. The lemma, script≡rattle-unchecked, is stated formally in Figure 5, and says:

> *For all builds, b, where Cmds in b do not write to their reads, executing b with rattle-unchecked produces a FileSystem equivalent to the one produced by executing b with script.*

This is analogous to the correctness of Fabricate in §4, but notably does *not* include a requirement that the build be hazard free. This is because rattle-unchecked records both the files a command read and wrote to, and will re-run a command if the reads or writes of a command have changed since it was last run, rather than just the reads. Therefore, there is no need to prove the writes of a command haven't changed since it was last run. Evidence is still required that commands do not write to their own inputs, which is provided by DisjointBuild, because we need to know the values of the files read by a command are accurately recorded in the Memory, since those values are recorded after a command has *completed*.

5.3 Correctness of Sequential Rattle

We now wish to prove the *correctness* lemma for sequential Rattle. Following the correctness definition for a forward build system stated in §2.3, we provide the following informal correctness definition for sequential Rattle:

> *Sequential Rattle is correct if running a build, b with Rattle, either results in a hazard or a file system equivalent to the one produced by running b as a script.*

The formal correctness lemma in Agda, correct-rattle is in figure 6.

This definition of correctness is faithful to the one in §2.3, because the only hazards in a sequential build, are those

```
script≡rattle-unchecked : ∀ s b → DisjointBuild s b → (∀ f₁ → script b s f₁ ≡ proj₁ (rattle-unchecked b (s , [])) f₁)
```

Figure 5. A lemma stating rattle-unchecked and script produce equivalent FileSystems.

```
≡toScript : FileSystem → Build → Build → Set
≡toScript s br bs = ∃[ s₁ ](∃[ m ](∃[ ls ](rattle br bs ((s , []) , []) ≡ inj₂ ((s₁ , m) , ls) × ∀ f₁ → s₁ f₁ ≡ script bs s f₁)))

correct-rattle : ∀ s b → PreCond s b b → ¬ HazardFree s b b [] ⊎ ≡toScript s b b

soundness : ∀ {s₁} {m₁} {ls} s br bs → DisjointBuild s br → rattle br bs ((s , []) , []) ≡ inj₂ ((s₁ , m₁) , ls)
          → (∀ f₁ → script br s f₁ ≡ s₁ f₁)

completeness : ∀ s br bs → PreCond s br bs → HazardFree s br bs [] → ∃[ st ](∃[ ls ](rattle br bs ((s , []) , []) ≡ inj₂ (st , ls)))
```

Figure 6. The correctness lemma for sequential Rattle, correct-rattle. As well as the soundness and completeness lemmas used to prove it.

caused by problematic sequences of commands in the build script and not due to speculation. In the remainder of this section we prove soundness and completeness before finally proving the correctness of sequential Rattle.

5.3.1 Soundness.

Rattle is sound if it preserves the semantics of the Script. Applied to our Agda model of Rattle, rattle is sound if when it does not find a Hazard, the resulting FileSystem is equivalent to the one produced by script. In Figure 6 we formally state a soundness lemma for rattle:

> *rattle is sound if, when executing a Build, br, if it produces a FileSystem and not a Hazard, then script produces an equivalent FileSystem when executing br.*

We omit the details of the proof here, but rattle is sound because runWError is sound, meaning the FileSystem runWError produces is the same one runR produces. The proof of runWError's soundness is trivial, so we also omit the details here.

5.3.2 Completeness.

The final lemma we introduce and prove is the completeness of rattle. Rattle is complete if for all builds with no hazards it does not discover a hazard. We state a corresponding lemma in our Agda model for rattle in Figure 6:

> *rattle is complete if for any Build, br, where the standard preconditions are true, and which is HazardFree, executing br with rattle produces a FileSystem and Memory, and not a Hazard.*

At a high level, the proof of *completeness* shows for each Cmd in *br*, that runWError produces a new FileSystem and Memory and not a Hazard, because it would be a contradiction to produce a Hazard. Therefore, rattle will produce a FileSystem and Memory rather than a Hazard.

5.3.3 Correctness.

The proof of *correctness*, correct-rattle in figure 6, easily follows from *soundness* and *completeness*.

6 Correctness of Speculative Rattle

In the previous section we presented a correctness definition for sequential Rattle and proved sequential Rattle is correct. In this section we present a correctness definition for Rattle with speculation, where Rattle is executing an ordering of commands other than the one specified by the build author, and prove in its current state that Rattle is only partly correct.

Rattle achieves parallelism by *speculating* commands before they are *required* by the build script. A command is *required* when the build script passes it to the build system with a call to `cmd`. If Rattle already speculated the command it can skip it and immediately continue. Rattle uses the commands the build *required* last time it was executed to decide which commands to speculate. Things can go wrong when running a build with speculation. First, if the build script has changed since it was last run Rattle could speculate commands that are no longer part of the build. Second, the dependencies of the recorded commands could have changed since they were last run causing Rattle to speculate commands when it shouldn't have. We limit ourselves to proving what happens in the case where Rattle only runs commands in the build, but we will briefly discuss how we could extend the lemmas for the case where Rattle runs unnecessary commands in §6.3.

Rather than encode the speculation algorithm used by Rattle, we instead model two builds, the *script* build, *bs*, which is the one the user wants to execute and the *speculative* build, *br*, which is the build Rattle actually executes. The total correctness lemma and the partial correctness lemma's we state below are for all builds, where *br*

```
correct-speculation : ∀ s br bc → PreCond s br bc → ¬ HazardFree s bc bc [] ⊎ ≡toScript s br bc
```

Figure 7. A total correctness lemma for Rattle with speculation, which says the script build has a hazard, or running the re-ordered build has the same effect as running script build. We *cannot* prove this lemma for the current implementation of Rattle.

is a permutation of *bs*, and thus regardless of the strategy Rattle chooses for speculating builds, our model and lemmas are correct. These proofs leave Rattle free to speculate any way it chooses, balancing performance against the likelihood of hazards.

6.1 Total Correctness (Not Provable)

We first state a total correctness lemma for Rattle with speculation, which we cannot prove:

> *Speculative Rattle is correct if either the script build contains a* read before write *or* write before write *hazard, or executing the speculative build with Rattle is equivalent to executing the script build as a script.*

Informally, if the script build has no *read before write* or *write before write* hazards, then whatever speculatively reordered build Rattle decides to run produces a result equivalent to executing the script build. We state this lemma in Agda, correct-speculation in Figure 7, but are not able to prove it.

In practice, Rattle first runs a build with speculation, and if a hazard occurs, reruns the build without speculation in the hope the hazard was a consequence of speculation rather than inherent in the build. However, it is possible in rare circumstances that speculation might mess up the *inputs* to the build (probably because in previous runs they were outputs). As a result, there might be builds that have hazards caused by speculation, but no real hazards, and cannot be executed equivalent to a script. There are techniques discussed in the Rattle paper [14] to address these (using `git` for inputs, segregating inputs from outputs), but none are yet implemented in the Rattle tool. If Rattle were to address this issue, we believe Rattle could be proven to be correct by the above definition, but as it is, we seek to prove partial correctness of Rattle.

6.2 Partial Correctness

Our partial correctness theorem weakens the total correctness theorem with an additional clause:

> *Speculative Rattle is partially correct if either the script build contains a* read before write *or* write before write *hazard, or the speculative build contains a hazard, or executing the speculative build with Rattle is equivalent to executing the script build as a script.*

Specifically, we consider a hazard caused by speculation to be a partially correct execution. See Figure 9 for the *partial correctness* lemma and proof in Agda, *semi-correct*.

Before we prove *partial correctness* we require an additional lemma stated below, and in Agda in figure 8:

> *For two builds br and bs, which are permutations, where br is HazardFree with respect to bs, executing br with script produces a FileSystem equivalent to the one produced by executing bs with script.*

We prove this *reordering* lemma (reordered≡ in Figure 8) by assuming it is true for *br* and *bs* with *x*, the last Cmd in *bs* removed from both. And then showing that adding *x* back to both *br* and *bs* still results in equivalent FileSystems because *x* reads and writes to the same files when run as part of *br* and *bs*, and *x* does not write to any file another Cmd in either *br* or *bs* read or wrote to, because if it did there would be a Speculative hazard.

We can now explain our proof of *partial correctness*. Using the decidability of HazardFree, we can case on whether or not the speculative build, *br* is HazardFree with respect to *bs*. If it is HazardFree, we can trivially use soundness, completeness and reordered≡ to show that rattle executing *br* is equivalent to script executing *bs*.

6.3 Correctness with Extra Commands

The lemmas in this section state correctness if Rattle executes a permutation of the script build, but not what happens when Rattle introduces extra commands. When Rattle is executing unnecessary commands we consider the "output" of the build to be those files written to by the script build. We would then say executing a build with extra commands is equivalent to executing the script build, if the files written to by the script build have the same values in both FileSystems. All of our lemmas could be adjusted to support this definition of correctness by stating two FileSystems are equivalent for the set of files written to by the script build rather than equivalent for all files. Future work involves proving these alternative correctness lemmas.

6.4 Bug Detecting *speculative write before read* Hazards

Speculative write read hazards occur when a command required by the build script, reads a file, which a *speculated* command has already written to. The implementation of

```
reordered≡ : ∀ s br bc → PreCond s br bc → HazardFree s br bc [] → (∀ f₁ → script bc s f₁ ≡ script br s f₁)
```

Figure 8. The reordering lemma for script, which says for two builds which are permutations of one another, and are HazardFree, executing both with script is equivalent.

```
semi-correct : ∀ s br bs → PreCond s br bs → ¬ HazardFree s br bs [] ⊎ ¬ HazardFree s bs bs [] ⊎ ≡toScript s br bs
semi-correct s br bs pc with hazardfree? s br bs []
... | no hz = inj₁ hz
... | yes hf₁ with completeness s br bs pc hf₁
... | (s₁ , m₁) , ls , ≡₁ = inj₂ (inj₂ (s₁ , m₁ , ls , ≡₁ , ∀≡))
  where ∀≡ : ∀ f₁ → s₁ f₁ ≡ script bs s f₁
        ∀≡ f₁ = sym (trans (reordered≡ s br bs pc hf₁ f₁)
                           (soundness s br bs (proj₁ pc) ≡₁ f₁))
```

Figure 9. The partial correctness lemma for speculative Rattle; which says *br* has a hazard, or *bs* has a hazard or running *br* has the same effect as running *bs*.

Rattle currently checks for hazards when a command finishes running, and in order to provide an efficient implementation (hazard checking can be quite expensive) only considers the files that were read or written by that command. Moreover, when a command is required but skipped because it was already run, Rattle does not recheck for hazards. As a consequence, there is a bug that if the command, x, was marked as *speculated* when it went through hazard checking, but later become *required*, Rattle will potentially fail to detect a *speculative write read* hazard involving x.

When attempting to prove and model Rattle in Agda it became obvious that if the commands were only considered when they were run, some speculative write read hazards would be missed. In particular, a required? predicate needed to be tested with the full set of commands, not a prefix, as Rattle was doing. The fix in the model was to see if the command is ever required, and the fix in Rattle would be to retest for hazards when a command changes required status.

This flaw in Rattle was far from obvious, and the tension between efficient implementation and simplicity made it hard to spot. Testing for the correctness properties is hard, especially around speculative hazards, as Rattle is quite careful as to what it speculates – making it hard to find a sequence of edits that would cause a problem with a speculative hazard. In contrast, the approach of modeling speculation as producing all permutations makes the bug trivial to find with Agda (or any proof-based technique).

7 Related Work

This paper formalises forward build systems and proves certain properties about these systems. While forward build systems have been around a long time, they are undoubtedly less popular than the more traditional backward build systems. Backward build systems were classified in the Build Systems à la Carte paper [11], which contains small representative implementations for each type of build system, along with definitions of correctness (that the result is equivalent to rebuilding everything) and minimality (only actions whose dependencies have changed are run). However, the implementations were in Haskell, and the definitions were specified informally. Those definitions were subsequently formalised [8], using Coq to reimplement the build systems. While working towards correctness and minimality (not yet achieved when that paper was published) it was determined that the notion of acyclic tasks (tasks which do not depend on themselves, including indirectly) was an important constraint to prove termination.

In addition to the formalisation of generic build systems, we are aware of two specific backwards build systems that have been had properties proven about them in more depth – specifically the systems Pluto and CloudMake. Starting with Pluto, the paper introducing Pluto [3] included a simplified rebuild algorithm (without cycle support), and then provided manual proofs of soundness (what we call correctness) and optimality (also known as minimality). Compared to our system, soundness property S3 most closely matches our definition, with the property that everything that was built must now be up to date. For their minimality property, the proof relies on a cache so nothing will be built twice (which Rattle shares) alongside a proof that nothing not required is built. For Rattle, because speculation might introduce unnecessary work, we do not expect minimality under these definitions. These proofs were reused and extended to apply them to model-driven development [15], showing the utility of these proofs. We are aware of an existing project to mechanise these Pluto proofs[3] and eagerly await the results.

[3] https://wp.doc.ic.ac.uk/vetssannualreport/mechanising-the-theory-of-build-systems/

The CloudMake build system was modeled formally by Christakis et al. [1] using Dafny [6]. A CloudMake build is driven by a script in a mutation-free subset of JavaScript, which is then modeled in Dafny, along with the rebuild algorithm. In particular there is an operation exec that executes an external program, with specific axiomatized properties, which has similarities our `Cmd` definition. The authors focus is not on proving that the build is correct, but proving that certain optimisations performed by CloudMake do not change the semantics. Using the framework the authors are able to prove that functions can be evaluated in parallel and that operations are able to be cached, giving the key properties of parallelism and incrementality as an optimisation over the standard build algorithm.

In this paper we have focused on the Rattle build system, but also included Memoize and Fabricate, which can be viewed as a subset of Rattle without speculation. There are other forward build systems, which contain different features that Rattle does not. Both Fac [13] and Stroll [10] do not require a complete order to be given, meaning there is no corresponding Shell evaluation order that can be matched. The LaForge [2] system uses tracing to observe execution at a more fine grained level than the user specifies, meaning that commands are sometimes executed in part. A command can be broken into sub-commands, which can be executed in place of the complete command. We believe the our high-level definition of correctness for forward build systems is applicable to these systems, but more work is needed to determine how to model them and prove that they meet this definition. For example, LaForge's partial execution of commands might result in additional hazards in typical cases.

8 Conclusion

Compilers, operating systems, and web servers need to be correct, and substantial effort has gone into verifying them. But a humble build system can sit in between all of these,[4] and thus must be correct as well. We modeled Rattle, a state-of-the-art forward build system, and proved it correct. In doing so we found a bug which we have a proposed fix for in Rattle.

While we have used this model to figure out if Rattle as it stands is correct, we haven't yet talked about what corrective action Rattle could take to recover from hazards. For *speculative* hazards, Rattle's current strategy is to rerun without speculation and thus without parallelism, which can be expensive. The reason for this simplistic strategy is that it's not yet clear what would be both better and correct. Our formal model will let us figure that out formally in Agda initially, before transferring the knowledge to Rattle itself.

[4] https://www.fireeye.com/blog/threat-research/2020/12/evasive-attacker-leverages-solarwinds-supply-chain-compromises-with-sunburst-backdoor.html

References

[1] Maria Christakis, Rustan Leino, and Wolfram Schulte. 2014. Formalizing and Verifying a Modern Build Language. In *FM 2014: Proceedings of Formal Methods (Lecture Notes in Computer Science, Vol. 8442)*. Springer, 643–657.

[2] Charlie Curtsinger and Daniel W. Barowy. 2021. LaForge: Always-Correct and Fast Incremental Builds from Simple Specifications. (2021). https://arxiv.org/abs/2108.12469.

[3] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A Sound and Optimal Incremental Build System with Dynamic Dependencies. In *Proceedings of OOPSLA 2015* (Pittsburgh, PA, USA). ACM, 89–106. https://doi.org/10.1145/2814270.2814316

[4] Stuart Feldman. 1979. Make - A program for maintaining computer programs. *Software: Practice and experience* 9, 4 (1979), 255–265.

[5] Berwyn Hoyt, Bryan Hoyt, and Ben Hoyt. 2009. Fabricate: The better build tool. (2009). https://github.com/SimonAlfie/fabricate.

[6] K Rustan M Leino. 2010. Dafny: An Automatic Program Verifier for Functional Correctness. In *LPAR 2010: Logic for Programming, Artificial Intelligence, and Reasoning*. 348–370. https://doi.org/10.1007/978-3-642-17511-4_20

[7] Nándor Licker and Andrew Rice. 2019. Detecting incorrect build rules. In *Proceedings of ICSE 2019*. ACM, 1234–1244. https://doi.org/10.1109/ICSE.2019.00125

[8] Georgy Lukyanov and Andrey Mokhov. 2019. Towards a Coq Formalisation of Build Systems. In *The Fifth International Workshop on Coq for Programming Languages*.

[9] Bill McCloskey. 2008. Memoize. (2008). https://github.com/kgaughan/memoize.py.

[10] Andrey Mokhov. 2021. Stroll: a build system that doesn't require a plan. In *IFL 21: Implementation and Application of Functional Languages*.

[11] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. 2020. Build systems à la carte: Theory and practice. *Journal of Functional Programming* 30, 55. https://doi.org/10.1017/S0956796820000088

[12] Ulf Norell. 2008. Dependently Typed Programming in Agda. In *Proceedings of the 6th International Conference on Advanced Functional Programming* (Heijen, The Netherlands) *(AFP'08)*. Springer-Verlag, Berlin, Heidelberg, 230–266.

[13] David Roundy. 2019. Fac build system. (2019). https://sites.science.oregonstate.edu/~roundyd/fac/.

[14] Sarah Spall, Neil Mitchell, and Sam Tobin-Hochstadt. 2020. Build Scripts with Perfect Dependencies. In *Proceedings of the ACM Programming Languages 4, OOPSLA*. Article 169, 28 pages. https://doi.org/10.1145/3428237

[15] Perdita Stevens. 2020. Connecting software build with maintaining consistency between models: towards sound, optimal, and flexible building from megamodels. *Software and Systems Modeling* 19, 4 (2020).