A Sound and Optimal Incremental Build System with Dynamic Dependencies

Sebastian Erdweg Moritz Lichter Manuel Weiel

TU Darmstadt, Germany

Abstract

Build systems are used in all but the smallest software projects to invoke the right build tools on the right files in the right order. A build system must be sound (after a build, generated files consistently reflect the latest source files) and efficient (recheck and rebuild as few build units as possible). Contemporary build systems provide limited efficiency because they lack support for expressing fine-grained file dependencies.

We present a build system called *pluto* that supports the definition of reusable, parameterized, interconnected builders. When run, a builder notifies the build system about dynamically required and produced files as well as about other builders whose results are needed. To support fine-grained file dependencies, we generalize the traditional notion of time stamps to allow builders to declare their actual requirements on a file's content. *pluto* collects the requirements and products of a builder with their stamps in a build summary. This enables *pluto* to provides provably sound and optimal incremental rebuilding. To support dynamic dependencies, our rebuild algorithm interleaves dependency analysis and builder execution and enforces invariants on the dependency graph through a dynamic analysis. We have developed *pluto* as a Java API and used it to implement more than 25 builders. We describe our experience with migrating a larger Ant build script to *pluto* and compare the respective build times.

Categories and Subject Descriptors D.2.9 [*Management*]: Software configuration management; D.2.11 [*Software Architectures*]: Languages

Keywords *pluto*; build system; builder API; incremental building; dynamic dependencies; cyclic dependencies

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3689-5/15/10...$15.00
http://dx.doi.org/10.1145/2814270.2814316

1. Introduction

Software developers struggle with build systems on a regular basis. Previous studies show that on average 12% of development effort is not spent on developing software but on maintaining build scripts [11]. Another study finds that build scripts in existing software projects continuously change and grow in complexity in sync with changes to the project's source code [12]. These studies show that the development and maintenance of build scripts is an essential part of software development.

As with any software artifact, build scripts are difficult to implement correctly. This is particularly true because users want build scripts to run fast enough to provide interactive feedback [10]. Existing build systems such as Make or Ant try to address this requirement through support for incremental rebuilding: After a file changes, the build system identifies and executes only those build operations whose result has been invalidated by the change. Unfortunately, existing build systems do not support sufficiently fine-grained file dependencies that are necessary for supporting sound and optimal incremental rebuilding. Instead, existing build systems frequently require clean builds (`make clean && make all`) in order to work correctly.

We propose a novel build system called *pluto* that features reusable, parameterized, interconnected builders. A builder can execute any number of build operations, read and write files, and trigger other builders. *pluto* tracks the dependencies of a builder dynamically and organizes them in a single dependency graph. To enable *sound* incremental rebuilding, *pluto* interleaves dependency analysis and builder execution and uses a dynamic analysis to enforce invariants on the dependency graph, such as builder A must trigger builder B before A may read any file generated by B. These invariants are essential to correctly determine in which order to execute builders after a file changed and whether a builder can be skipped. We provide a formal model of *pluto*'s dependency graph and prove the soundness and optimality of *pluto*'s rebuild algorithm relative to the previous dependency graph.

Since our rebuild algorithm is optimal, incrementalization is only limited by the precision of the builder-induced dependency graph. While existing build systems often over-

approximate dependencies, *pluto* aims to maximize incrementality by providing four mechanisms for fine-grained dependency tracking. First, *pluto* builders register file dependencies dynamically at build time. Second, builders can choose custom file stamps in place of the last-modified time to precisely declare which part of a file they depend on. Third, when running a builder, *pluto* automatically injects a dependency on the builder's implementation such that a change to the builder's implementation invalidates the builder's result. Fourth, *pluto* supports cyclic builder dependencies and allows custom cycle-handling strategies.

We have developed *pluto* as a Java API[1] and used it to realize builders for Latex, Bibtex, Java, and 23 other builders that we migrated from an existing Ant build script developed by others for the Spoofax language workbench [9]. For Spoofax, we measured *pluto*'s build times and found substantial speedups. To demonstrate the broad applicability of *pluto*, we furthermore used it to realize separate compilation and incremental rebuilding within two compilers. In summary, we make the following contributions.

- We review existing build systems with respect to their dependency granularity and incrementality (Section 2).
- We describe a novel build system *pluto* with fine-grained dependencies and demonstrate how it enables a builder for Latex with exact dependencies (Section 3).
- We formalize *pluto*'s dependency model and stipulate what it means for a build system to be sound and optimal (Section 4).
- We present *pluto*'s rebuild algorithm and verify its soundness and optimality (Section 5).
- We evaluate *pluto*'s applicability and its support for incremental rebuilds (Section 6).

2. Existing Build Systems

To clarify the need for a new build system, we review a few existing build systems with a focus on what kind of dependencies can be declared and how the build system supports incremental rebuilding.

GNU Make. A Make [16] build script consists of build rules of the form $\langle \mathit{Provide} : \mathit{Require}^* \ \mathit{Command}^* \rangle$. In Make terminology, the provided and required entities are called *targets*, which either refer to file names or to build-rule names. Make executes rules in a demand-driven fashion. When a target T is demanded, Make finds a rule that provides T, recursively demands the rule's required targets, and executes the rule's commands if any of the required targets was changed/rerun more recently than T. Thus, based on file and build-rule dependencies, Make provides a simple form of incremental building.

[1] Source code is online `https://github.com/pluto-build/pluto`

The problem of Make is that all file and build dependencies must be declared statically as part of a build rule. As consequence, build rules often have to approximate their dependencies, as a missing dependency makes the build script unsound: A rule fails to execute despite a change to a file that is actually needed. On the other hand, an overapproximation of dependencies limits incremental rebuilding: A rule executes because a file changed that is actually not needed. Neagu outlines the deficiencies of Make in greater detail in his article *What is wrong with make* [14].

Most build systems used in practice besides Make (e.g., MSBuild, Ninja, SCons, CloudMake) manage dependencies in a way that is very similar to Make. We discuss some notable exceptions below.

Shake. Similar to Make, a Shake [13] build script consists of build rules that statically declare all files they provide. However, unlike Make, Shake build rules can discover and register required files during building. That is, the dependency graph can change during run time of a build script. This makes many dependency patterns easy to express and allows Shake to support incremental building for them.

A common example for dynamic dependencies is the set of header files required to compile a C source file. The set of header files is platform-dependent and thus cannot be inlined into the build script. With Shake, a build script can simply call the C preprocessor as part of the build routine and extract the required header files from its output.

Shake is realized as a Haskell API and a build script takes the following form:

```
main = shakeArgs shakeOptions $ do
  Provide+ &%> \out -> do
    need Require*
    Command+
    need Require*
    Command+
    ...
```

Function *need* is part of Shake's API and registers a file dependency. As argument, *need* takes a file's path as a string, which can be, for example, computed from a command's output.

While Shake eliminates some overapproximation of dependencies, there is plenty of room for improvement. In particular, Shake provides non-optimal incrementalization because a build rule is considered outdated as soon as the last-modified time of a required file changes (independent of the file's content), because a clean build must follow any change of the build script itself, and because the files provided by a build rule need to be statically declared and cannot be computed. A simple example where the provided files are hard to determine statically is the compilation of Java source files: The compiler produces a class file for the main compilation unit and one class for each inner or anonymous class inside the compilation unit.

Apache Ant. An Ant[2] build script is an XML file that defines build rules of the following form:

```
<target name=Provide depends=Require* unless=Property>
    Command*
</target>
```

Unlike Make and Shake, Ant build rules do not express file requirements directly. Instead, an Ant build rule provides a name and requires the execution of other build rules by reference to their name. Thus, Ant's built-in dependency management does not provide any support for incrementality.

To support incremental rebuilding, an Ant build rule can declare its execution as being conditional using the attribute unless. The value of this attribute is the name of a global variable (*property* in Ant terminology). Ant skips a build rule if the variable referred to by unless is defined to be true. Since an Ant build script can set variables at any time, the skipping of a build rule is programmable in Ant.

For example, to incrementally rebuild targets based on file changes, Ant provides a generic macro *uptodate* to determine whether a required file changed more recently than a generated file. If the generated file is up-to-date, the macro sets a user-supplied variable, which can be used to skip the execution of a build rule:

```
<uptodate property="ok"
          srcfile="A.java" targetfile="A.class"/>
<target name="compile" unless="ok">...</target>
```

This mechanism is very flexible as it allows comparing the time stamps of arbitrary files. More generally, it is even possible to dynamically determine the relevant files for comparison or to incorporate other operations for deciding whether to skip a build rule. However, since the skipping of a build rule is programmable, this approach largely relies on the discipline of the build-script developer and, in general, is not sound. Moreover, full incrementality requires excessive use of *uptodate*, but even projects like Tomcat use *uptodate* only sporadically and require non-incremental rebuilds instead.

Problem statement. We provided a rough overview of dependency models and incremental building supported by existing build systems. We have seen dependencies of the following kind:

- build rule provides file (statically declared)
- build rule requires file (statically declared)
- build rule requires file (dynamically discovered)
- build rule requires build rule (statically declared)

The goal of *pluto* is to achieve optimal incremental rebuilding through maximally precise dependencies while also guaranteeing soundness. To this end, *pluto* should allow all dependencies of a build rule to be dynamically discovered and support advanced dependency mechanisms that further reduce the overapproximation of dependency declarations:

[2] http://ant.apache.org/

```
public abstract class Builder<In, Out> {
  protected abstract String description(In input);
  protected abstract File persistentPath(In input);
  protected abstract Out build(In input) throws Throwable;

  protected void require(File p, Stamper s) { ... }
  protected void provide(File p, Stamper s) { ... }
  protected <In_, Out_> Out_ requireBuild
      (BuilderFactory<In_, Out_> fact, In_ in) { ... }

  protected CycleHandlerFactory getCycleHandler() { ...}
}
```

Figure 1. Abstract builder underlying all *pluto* builders.

- User-provided file stamps that declare a dependency on parts of a file. To allow different build rules to apply different usage patterns, the stamp should be provided by the client and not by the producer of a file.
- When the definition of a build rule is changed, any previous result of this build rule must be discarded and rebuilt. Existing build systems fail to notice this and require the user to initiate a clean build instead.
- When recursive build-rule dependencies occur, existing build systems abort the build and indicate an error. Instead, it should be possible to provide dedicated strategies for handling cyclic dependencies.

In the following section, we illustrate *pluto* on a practical use-case that requires support for all of these goals.

3. Building with *pluto*

In this section, we describe *pluto*'s builder API and show how it enables the declaration of complicated dependency patterns, using a builder for Latex and Bibtex as our example.

3.1 The Builder API

pluto provides a Java API for implementing custom build scripts. A *pluto* build script consists of a collection of parameterized and interconnected builders. Figure 1 shows a slightly simplified version of the abstract builder that underlies all user-defined *pluto* builders.

A *pluto* builder takes input of type In and optionally produces output of type Out. To eliminate any accidental interactions between individual runs of a builder, we create a new builder instance for every build.

A builder must implement three abstract methods. First, a builder provides a description for logging purposes. Second, a builder provides a file path where the build summary is stored by *pluto*. This build summary remembers the required and provided files of the previous run and is essential for incrementalizing builds. Finally, a builder implements its build procedure in method build. Since the build procedure consists of regular Java code, a builder can make use of existing Java libraries and is easy to debug.

During the execution of the build procedure, a builder can register required and provided files via methods require and provide. Importantly, these methods not only take a file path but also a file stamper, which provides a single method to obtain a file stamp from a file path. The file stamp governs when a file is considered up-to-date. *pluto* provides a few generic stampers for builders to select from, such as last-modified time, file exists, file hash, and file content. However, a builder can also define and use its own domain-specific stamper, which can extract some information from the file and ignore the rest, such that only relevant file changes trigger a rebuild. However, it is important to note that different stampers have different performance and precision characteristics. For example, computing a file hash is more precise than comparing the last-modified times, but it also requires significantly more computation time. To save time, we tend to use the last-modified stamper for larger binary files and files that rarely change.

A builder can request the execution of another builder using method requireBuild. This method takes a builder factory, which constructs a fresh builder instance for the given input. *pluto* automatically consolidates the required build's build summary to determine if the build can be skipped. Finally, builders can request each other mutually recursively, thus forming a dependency cycle. Since different builders require different strategies for resolving cycles, a builder can override method getCycleHandler to provide a cycle handler. We provide details about cycle handling in Section 3.4.

3.2 A Builder for Latex and Bibtex

We illustrate *pluto*'s builder API by describing a builder for Latex documents. As input, our Latex builder takes the name of the main *tex* file as well as a source and target directory:

```
public class LatexInput {
  public final String docName;
  public final File srcDir;
  public final File targetDir;
  // initializing constructor
}
```

As output, our Latex builder provides the file path of the generated *pdf* document.

Figure 2 shows a builder for Latex documents with exact file dependencies. The builder description and persistent file path make use of the Latex input. Specifically, after a run of the Latex builder, *pluto* will store its build summary in file *latex.dep* in the target directory. *pluto* uses this build summary to determine whether a build is up-to-date. In principle, *pluto* could assign a persistent path automatically.

Method build does the actual building. As first step, the Latex builder requires a bibliography and requests the execution of the Bibtex builder. The Bibtex builder takes the same input as the Latex builder, but, in general, builders can be required with arbitrary input. The request to the Bibtex builder returns immediately if the Bibtex builder is up-to-date and otherwise blocks until the Bibtex build is finished.

```
public class Latex extends Builder<LatexInput, File> {
  public static BuilderFactory<LatexInput, File> factory =...;
  public static final CycleHandlerFactory latexBibtexCycle =
    FixpointCycleHandler.of(Bibtex.factory, Latex.factory);

  protected String description(LatexInput input) {
    return "Build PDF for " + input.docName;
  }
  protected File persistentPath(LatexInput input) {
    return new File(input.targetDir, "latex.dep");
  }
  protected CycleHandlerFactory getCycleHandler() {
    return latexBibtexCycle;
  }
  protected File build(LatexInput input) throws Throwable {
    requireBuild(Bibtex.factory, input);

    String docName = input.docName;
    File tex = new File(input.srcDir, docName + ".tex");
    File aux = new File(input.targetDir, docName + ".aux");
    require(tex, FileHash.instance);
    require(aux, FileHash.instance);

    ExecutionResult res = Exec.run(input.srcDir, "pdflatex",
      "-output-directory=" + input.targetDir,
      "-interaction=batchmode",
       "-kpathsea-debug=4",
       docName + ".tex");
    for (File p : extractReadFiles(res.errLog))
      if (!p.equals(tex) && !p.equals(aux))
        require(p, LastModified.instance);
    for (File p : extractWrittenFiles(res.errLog))
      provide(p, LastModified.instance);

    return input.targetDir.resolve(docName + ".pdf");
  }
}
```

Figure 2. A Latex builder with exact file dependencies.

As second step, the Latex builder identifies the main *tex* file in the source directory and the *aux* file in the target directory. The *aux* file is used internally by *pdflatex* to keep track of references and other information within the Latex document. The Latex builder requires both files using their file hash as a stamp, such that only a true edit of a file triggers a rebuild. This is beneficial for source files like our Latex document because it makes the build invariant under file-system operations (e.g., by version-control systems) that do not change the content but the last-modified time of a file. For the *aux* file, there is another reason for using a file hash as a stamp, namely Latex requires repeated building until the content of the *aux* file stabilizes (details in Section 3.4).

As third step, the Latex builder executes *pdflatex* with appropriate command-line arguments. Specifically, the Latex

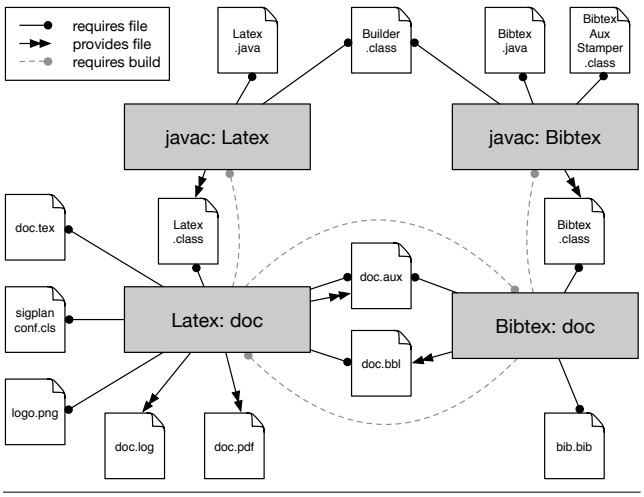

Figure 3. Exact dependencies of a simple Latex build.

builder uses the option `-kpathsea-debug=4`, which makes *pdflatex* print the path of each read and written file to stderr.[3] Our method Exec.run starts *pdflatex* in a new process with input.srcDir as working directory. Exec.run yields an ExecutionResult that contains all messages the command issued to stdout and stderr. The Latex builder parses these messages to extract the read and written file paths and registers required and provided files accordingly. As final step, the builder returns the file path of the generated *pdf* file, which is among the files written by *pdflatex*.

The Bibtex builder (not shown) is similar to the Latex builder, but calls *bibtex* instead of *pdflatex*. We want to emphasize two notable properties of the Bibtex builder. First, the Bibtex builder not only requires all involved bibliographies but also the *aux* file, generated by the Latex builder. To make sure the *aux* exists, the Bibtex builder requests the execution of the Latex builder, effectively forming a build cycle, which *pluto* specifically supports. The Latex and Bibtex builders explicitly define support for handling cycles (details in Section 3.4). Second, while the Bibtex builder depends on the *aux* file, most of this file is irrelevant for the Bibtex builder. To this end, the Bibtex builder uses a custom stamper that parses the *aux* file, extracts the bibliography name and the set of referenced citation keys, and uses them as a stamp. Accordingly, the Bibtex builder only becomes outdated when a different bibliography is used or a different set of bibliography entries is cited in the *aux* file. For example, changes to regular text or to captions of figures do not lead to a rebuild of the Bibtex builder.

3.3 Discussion

pluto achieves maximally incremental rebuilds while retaining soundness by supporting precise dependencies. Figure 3 shows the dependencies of our Latex builder for a simple latex document *doc.tex* in *sigplanconf* style including a graphics *logo.png* and a bibliography *bib.bib*. Rebuilds are incremental. For example, *pluto* only reruns the Bibtex builder if *bib.bib* changes, a citation is added or removed from the document, or the implementation of the Bibtex builder in *Bibtex.java* changes. Below, we discuss the core properties of *pluto* builders using this example.

Dynamic dependencies. Often, static dependency detection is difficult. In the case of Latex where even parsing is undecidable [4], soundly detecting dependencies is a lost cause. Therefore, in *pluto*, we support the dynamic discovery of dependencies, which a builder can register at any time during a build using methods require and provide. For example, dynamic dependency discovery allowed us to identify the Latex class *sigplanconf.cls*, the included graphics file *logo.png*, the bibliography *doc.bbl*, and the generated output *doc.pdf* in Figure 3.

File stamps. In other build systems, when a builder requires a file, the builder is rerun whenever the file's last-modified time changes. However, often a builder is only interested in parts of a file and invariant to changes in other parts of a file. For example, a JavaDoc builder is invariant to changes to a method's body and only interested in method signatures and comments. Similarly, the Bibtex builder is invariant to changes in a Latex document that do not change the set of cited Bibtex entries. *pluto* supports custom file stamps such that a builder can declare its actual dependency on a file's content. Since different builders can require different parts of the same file, it is important that the client of a file selects the file stamp rather than the provider of a file. For example, the Latex builder requires the same *aux* file as the Bibtex builder, but the Latex builder is sensitive to any change in the file's content and not just to the bibliography references. Finally, note that sophisticated file stamps are more expensive to check than simple ones. For example, the last-modified time is much faster to compute than a file's hash. In practical applications, build-script developers need to trade a stamp's cost off against the potential incremental speedup.

Builder dependencies. In *pluto*, a builder not only requires files but also other builds. For example, the Latex builder requires the Bibtex builder to run. In fact, it is an error to require a generated file without first requiring the builder that generates the file. This is essential for the soundness of the build system: If the Latex builder did not require the Bibtex builder but reads the *bbl* file nonetheless, the build result is only consistent when the Bibtex builder happens to be up-to-date (e.g., no pending change of the bibliography file). The behavior depends on the overall system state and is nondeterministic from the build system's point of view. Builder dependencies ensure that requiring a generated file consistently succeeds. This is particularly important for build systems that support incrementalization (or parallelization), because they may change the order in which builders are

[3] `http://tug.org/texinfohtml/kpathsea.html`

executed. *pluto* enforces appropriate build requests through a dynamic analysis.

Metadependencies. Beyond changes to required or provided files, a build result can also become inconsistent when the code of its builder changed. While almost all existing build systems ignore this kind of dependency, *pluto* automatically injects appropriate dependencies into build results. To this end, Figure 3 also shows the build units that result from calling a Java builder on the code that implements the Latex and Bibtex builders. In our example, the Java builder generates an executable Latex builder *Latex.class* from its source file *Latex.java*, linking it against the abstract builder class from *pluto*'s builder API. When *pluto* executes a builder, it uses Java reflection to find the corresponding class file and adds it as a required file to the build result. Since the class file is generated, *pluto* also needs to identify the corresponding builder and add it as a builder dependency as described above. As consequence of this dependency setup, when a developer changes *Latex.java*, a previously generated *pdf* document is regenerated because (i) build result *Latex: doc* requires build result *javac: Latex*, which requires the changed file *Latex.java* and thus needs rebuilding and (ii) this rebuilding changes *Latex.class*, which is required by build result *Latex: doc*, which thus needs rebuilding itself. This way, *pluto* guarantees build results are consistent after each run.

Dependency injection. Consider a Latex file that includes a generated *png* graphics that, for example, was generated from evaluation data to display the data's distribution. Since the *png* file was generated, it may seem that the Latex builder needs to unconditionally use method requireBuild to require the builder of the *png* file. Such hard-coded dependency would be bad because the Latex builder would have to be changed whenever the dependencies change, for example, if the *png* file is no longer needed.

To avoid such hard-coded dependencies, *pluto* supports dependency injection. Build requests are first-class values in *pluto* and can be forwarded to a builder, such that this builder can execute the build requests and install corresponding dependencies. For the *png* example, the Latex builder can receive a list of build requests as part of its input (LatexInput above does not show this). A request to build a Latex document that includes a generated *png* graphics may then look as follows.

```
BuildRequest pngRequest =
  new BuildRequest(PNG.factory, new PNGInput(...));
File pdf = requireBuild(
  Latex.factory, new LatexInput(..., pngRequest));
```

3.4 API for Handling Build Cycles

Most build systems assume that builders do not run into a cycle. However, cyclic code dependencies are not uncommon in practice, especially within a single language. In our example, there are actually two cycles. First, the Latex builder is in a cycle by itself because it reads from and writes to the same *aux* file. Second, the Latex and Bibtex builders depend on each other because the Latex builder writes to the *aux* file that Bibtex reads and Bibtex writes to the *bbl* file that Latex reads. Without cycle support, we would have to merge the Latex and Bibtex builders into a single builder, which itself has to make sure that Latex runs often enough and that Bibtex is not called on every single run but only when actually needed.

```
public interface CycleHandler {
  public String cycleDescription(BuildCycle cycle);
  public boolean canBuildCycle(BuildCycle cycle);
  public Set<BuildUnit> buildCycle(BuildCycle cycle,
    BuildManager manager) throws Throwable;

}
```

Figure 4. Interface for cycle handlers.

Different builders require different strategies for building cycles. Therefore, our API allows the definition and selection of different cycle handlers. Figure 4 shows the definition of the interface common to all cycle handlers. Like for builders, we create fresh cycle handlers for each occurrence of a cycle to avoid accidental interactions. By default, a builder does not provide a cycle handler, which is the usual case in build systems. If a builder is designed to support cycles, it can provide a cycle handler by overriding method getCycleHandler from the builder API (Figure 1).

When *pluto* detects a build cycle, it queries all builders in the cycle for a cycle handler. If none of the builders provides a cycle handler, the cycle cannot be built and building is aborted. Otherwise, *pluto* asks each cycle handler whether it can build this particular cycle (method canBuildCycle). If at least one cycle handler can build the cycle, *pluto* invokes its method buildCycle. This method takes the build cycle and an instance of the currently running build manager. It produces a set of build units (cf. Section 4), one for each member of the cycle. Method buildCycle can call method build of the involved builders and needs to resolve the cyclic dependencies. We implemented two predefined generic cycle handlers that can be used by builders as needed.

Fixpoint cycle handler. This cycle handler invokes the builders in the cycle iteratively until they stabilize, that is, until a fixpoint is reached. In each iteration, the cycle handler checks which builders in the cycle need to be reinvoked because some dependencies are not up-to-date. The handler then invokes these builders. If no builder required invocation, the building reached a fixpoint and the cycle building stops. If at least one builder was invoked, the cycle handler starts the next iteration. It is important to note that the fixpoint cycle handler itself does not guarantee a fixpoint will be reached. The builders in the cycle need to be designed to take part in a fixpoint compilation. We implemented the fixpoint cycle handler independently of the builders that participate

P ::= <path> — path to a file
Ω ::= $P \to$ <file>$_\perp$ — file system

FS ::= $P \times \Omega \to S$ — file stamper
S ::= stamp <value> FS — file stamp

R ::= freq P S | breq B I — file or build requirement
G ::= gen P S — provided file with stamp
I ::= <value> — builder input
B ::= $\{build\colon I \times \Omega \to U \times \Omega,\ path\colon I \to P\}$ — builder
U ::= $\{builder\colon B,\ input\colon I,\ reqs\colon \overline{R},\ gens\colon \overline{G}\}$ build unit

Figure 5. Syntax of build units and requirements

in a cycle, which makes the handler reusable. We use the fixpoint cycle handler in the Latex and Bibtex builder from the example above.

Build-at-once cycle handler. This cycle handler collects all the input of the cyclic builders and builds them simultaneously by invoking a single builder on the collected input. Thus, the build-at-once cycle handler only supports cycles where all builders are equal and only differ with respect to their input. Moreover, the involved builder must provide a build method that accepts a set of inputs. In the noncyclic case, the builder is invoked with a singleton set. When a cycle is detected, the build-at-once handler invokes the builder on all these inputs. For example, we use this cycle handler in our Java builder. The builder takes multiple source files as input. If they do not depend on each other cyclicly, we invoke the java compiler on each source file separately. But if there is a cyclic dependency between files, we pass them to the Java compiler together.

4. A Formal Model of *pluto* Dependencies

The previous section introduced *pluto* from a user's perspective. In this section, we formally describe the dependency graph used by *pluto* for dependency management and how *pluto* manages metadependencies and cycles. On top of the dependency graph, we precisely define what it means for a build result to be up-to-date and what constitutes a sound and incremental build system.

4.1 A Two-Layered Dependency Graph

pluto represents dependencies in a two-layered dependency graph. The file layer of a dependency graph contains nodes that represent required and provided files. File nodes are never connected. The build layer of a dependency graph contains nodes that represent the result of a builder. We call such nodes *build units*. *pluto* connects build units to those nodes in the build layer that the unit required to be built. In addition, *pluto* connects a build unit to all nodes in the file layer that represent files required or provided by the build unit. Figure 3 shows an example dependency graph resulting from a Latex build.

Figure 5 defines the syntax of build units and requirements used by our dependency graph. Since build systems read and write files, we need to model a file system. We represent the file system as a function $\omega \in \Omega$ that retrieves a file handle <file> for a path $p \in P$ or $\perp$ if the file does not exist. We assume the existence of generic functions for reading ($\mathsf{read}_T : P \times \Omega \to T_\perp$) and writing ($\mathsf{write}_T : T \times P \times \Omega \to \Omega$) arbitrary data to the file system.

To model precise file dependencies, the edges in *pluto*'s dependency graph are labeled with file stamps $s \in S$. We model a file stamper FS as a function that takes the path of a file and yields a stamp. A stamp S consists of an arbitrary stamp value <value>. To allow up-to-date checks, a stamp stores a reference to the stamper that created the stamp (details in Section 4.2). A file requirement (freq p s) stores the path of the required file together with the file stamp for that path, as does a provided file (gen p s).

We represent builders $b \in B$ similar to our Java API as a record with fields $build$ and $path$. One notable difference to our Java API is that we left out the in-memory builder output in our formal model. We write $b.label$ to access the data stored at field $label$. A builder provides a function $build$ that implements the build procedure. The build procedure takes the builder's input $i \in I$ and yields a build unit $u \in U$. The build procedure may read and write any file from the file system and may trigger the execution of other builders. We assume that builders correctly register file dependencies in the resulting build unit. In addition to the build procedure, a builder also provides a function $path$ that yields the path for storing and retrieving the build unit resulting from invocations of $build$. A build unit then simply is a record collecting all relevant information: the builder, its input, the required files and required builds $reqs$, and the provided files $gens$. We use the helper function

$$\mathsf{path}(u) = u.builder.path(u.input).$$

We represent edges on build units through relation $requires \subset U \times \Omega \times U$, where $(u\ requires_\omega\ v)$ if and only if there are $b \in B$ and $i \in I$ such that (breq b i) $\in u.reqs$ and $\mathsf{read}_U(b.path(i), \omega) = v$. We write $requires_\omega^+$ for the transitive closure and $requires_\omega^*$ for the reflexive transitive closure of $requires_\omega$.

A two-layered *dependency graph* DG is a set of build units $DG \subset U$. A dependency graph is ω-well-formed if it adheres to the following conditions:

1. Closed under $requires_\omega$: If $u \in DG$ and $u\ requires_\omega\ v$, then $v \in DG$.
2. No overlapping build-unit files: If $u, v \in DG$ and $u \neq v$, then $path(u) \neq path(v)$.
3. No overlapping generated files: If $u, v \in DG$, $u \neq v$, and (gen p _) $\in u.gens$, then (gen p _) $\notin v.gens$.
4. No hidden dependencies: If $u, v \in DG$, $u \neq v$, $u.reqs = \langle r_1, \ldots, r_n \rangle$, $r_k =$ (freq p _), and (gen p _) $\in v.gens$,

then for some $j < k$, $r_j = (\mathsf{breq}\ b\ i)$ such that $\mathsf{read}_U(b.path(i), \omega) = x$ and $x\ requires^*_\omega\ v$.

The conditions ensure that changing the build order does not affect the build results. The first condition makes sure a dependency graph contains all required build units. The second condition checks that different builders provide distinct paths for storing their build unit. Similarly, the third condition ensures different builders do not generate the same files. A violation of any of the previous two conditions would make building sensitive to build order. The last condition ensures all file dependencies are governed by builder dependencies. That is, before a build unit u can require a file generated by another build unit v, u must (transitively) require v. Accordingly, our condition checks that, preceding the file requirement, there is a build requirement $(\mathsf{breq}\ b\ i)$ of some unit that requires v. This condition is most important as it guarantees that build-unit/build-unit dependencies soundly approximate build-unit/file dependencies. In our implementation, *pluto* checks all four conditions incrementally as requirements are discovered. In case of a violation, we abort the build and provide a meaningful error message.

4.2 File Stamps and Build-Unit Consistency

We consider a build unit to be consistent if all required build units are consistent as well as all required and generated files.

We use customizable file stamps to track file changes. Given a path p and file system ω, we consider a file to be *up-to-date* with respect to a stamp $(\mathsf{stamp}\ v\ f)$ if $f(p, \omega) = v$. Thus, only a change to p that leads to a change of its stamp $f(p, \omega)$ will invalidate a file requirement $(\mathsf{freq}\ p\ (\mathsf{stamp}\ v\ f))$ or generated file $(\mathsf{gen}\ p\ (\mathsf{stamp}\ v\ f))$.

For build-units, we distinguish internal from total consistency. A build unit u is *internally consistent* with respect to a file system ω if

1. all generated files $(\mathsf{gen}\ p\ s) \in u.gens$ are up-to-date,
2. all required files $(\mathsf{freq}\ p\ s) \in u.reqs$ are up-to-date, and
3. for all required builds $(\mathsf{breq}\ b\ i) \in u.reqs$, the corresponding build unit $v = \mathsf{read}_U(b.path(i), \omega)$ exists and satisfies $v.builder = b$ and $v.input = i$.

A build unit u is *totally consistent* (or just *consistent*) with respect to ω if all build units v with $u\ requires^*_\omega\ v$ are internally consistent. Thus, total consistency of u requires internal consistency of u itself and of all build units that u transitively depends on.

4.3 Sound Build Systems

A build unit requires rebuilding if it is not totally consistent. Conversely, a totally consistent build unit does not require rebuilding. It is the job of a build system to produce a set of totally consistent build units that do not require rebuilding. For soundness, it does not matter if the build system performs a clean build or incrementally rebuilds some build units.

We model a build system as a function $build : (\overline{B \times I}) \times \Omega \to (\overline{U})_\perp \times \Omega$. A build system takes a sequence of build requests, represented as pairs of builders and builder inputs, and a file system as input. In addition to a possibly updated file system, the build system yields an error $\perp$ if the executed builders violate any of the well-formedness conditions for dependency graphs. Otherwise, a build system yields one totally consistent build unit for each build request forming a well-formed dependency graph. Formally, a build system is *sound* if it satisfies the following soundness criteria:

(S1) If $build(\overline{(b, i)}, \omega_0) = (\overline{u}, \omega)$, then the closure of $\overline{u}$ under $requires_\omega$ is an ω-well-formed dependency graph.

(S2) If $build(\langle (b_1, i_1) \cdots (b_m, i_m) \rangle, \omega_0) = (\langle u_1 \cdots u_n \rangle, \omega)$, then $m = n$ and for all $x \in \{1, \ldots, m\}$, $u_x.builder = b_x$ and $u_x.input = i_x$.

(S3) If $build(\overline{(b, i)}, \omega_0) = (\overline{u}, \omega)$, then all $u \in \overline{u}$ are totally consistent with respect to ω.

The soundness of a build system can only hold if the involved builders satisfy a sanity check and yield reasonable results. Otherwise, a builder could always yield an inconsistent build unit, triggering a sound build system impossible.

Assumption 4.1. For any builder $b \in B$ with input $i \in I$, if $b.build(i, \omega_0) = (u, \omega)$, then $u.builder = b$, $u.input = i$, and u is internally consistent with respect to ω.

Note how our formalization assumes the build system has exclusive access to the file system. In practice, it is sufficient to ensure that no other process changes any files required or generated by any of the involved builders. Even if this cannot be ensured, a change from another process will at most contaminate the ongoing build; a subsequent build will detect the outdated file stamp and trigger a rebuild accordingly.

4.4 Incremental Build Systems

We consider incremental building at the level of build units. That is, if we execute a builder, we execute it completely. Therefore, our model of builders as a function from build input to build unit is sufficient. Users can realize incremental computations at different granularities by splitting or merging builders.

We call a sound build system *incremental* if it optimizes for the following two incrementality properties:

(I1) Minimize the number of builder-function executions.

(I2) Minimize the number of internal-consistency checks.

An incremental build system is *optimal* if, for any input, it executes the minimal number of builders (I1) and checks internal consistency as infrequently as possible (I2), where (I1) takes precedence over (I2). As a corollary, an optimal build system is idempotent when the file system is unchanged: If $build(\overline{(b, i)}, \omega_0) = (\overline{u}, \omega)$, then $build(\overline{(b, i)}, \omega) = (\overline{u}, \omega)$.

As a small caveat, a build system can only be optimal if the involved builders satisfy a stability condition. That

is, a builder may not drop a requirement if all previous requirements are up-to-date. Otherwise, a builder could drop requirements arbitrarily, making incremental rebuilding unpredictable and optimality impossible.

Assumption 4.2. For any builder $b \in B$ with input $i \in I$, let $b.build(i, \omega_0) = (u, \omega)$ and $u.reqs = \langle r_1, \ldots, r_k, \ldots, r_n \rangle$. If r_k is a build requirement and all $r_1, \ldots, r_{k-1}$ are build requirements or up-to-date file requirements in ω, then the rebuild $b.build(i, \omega) = (v, \omega')$ retains r_k, $r_k \in v.reqs$.

Based on this assumption, given a build-system invocation $build(\overline{(b, i)}, \omega_0) = (\overline{u}, \omega)$, a sound build system is optimal if it only rebuilds those units v reachable from $\overline{u}$ for which $b.path(i)$ is invalid in ω_0 or for which v is internally inconsistent after all of its build requirements have been made totally consistent.

4.5 Metabuilding: Building Builders

As discussed in Section 3.3, a build result also becomes inconsistent if the definition of the builder changes. So far, our formal model has ignored this issue.

Typically, building a builder amounts to generating one or more executable files and using these executable files as a builder. For the sake of simplicity, let us assume that building a builder generates a single file at path p that we can deserialize into a builder using $\mathsf{read}_B(p, \omega) = b$. When executing b, we automatically add a file requirement on p and a build requirement on the build unit u_b that generated p. Thus, when a file that is required for building u_b changes, u_b gets built again, which makes our file requirement on p outdated and triggers a rerun of b. It is important to notice that the build and file requirements that we add as metadependencies are regular requirements. For this reason, any sound build system will detect a change to the definition of b, trigger a rebuild of b, detect changes to the executable files generated for b, and trigger a rebuild of any result built by b.

In our implementation, when a builder b is executed, we use Java's reflection to find the *jar* or *class* file p that implements b. If the file was generated by another builder, we locate the corresponding metabuild unit u_b and add requirements on p and u_b to units produced by b. When *pluto* finds an outdated metadependency, it not only triggers a rebuild but also uses the JVM's hotswapping support to dynamically reload the rebuilt class files. If hotswapping fails (e.g., when adding or removing a field), the JVM needs to be restarted by the user in order to run the latest builder definition. *pluto* detects this situation and issues a corresponding error message.

4.6 Cyclic Dependencies

pluto allows build units to mutually recursively depend on each other. We represent cyclic dependencies in the same way as non-cyclic dependencies using a build unit's requirements field $reqs$. In particular, we call a build unit u cyclic in ω if $u \ requires^+_\omega \ u$. Cyclic build units adhere to the same definitions for internal and total consistency as their non-cyclic counterparts. However, according to the definition of total consistency, either all or none of the units involved in a cycle are totally consistent.

To build a cycle, it is not possible to simply execute all involved builders, because the builders require each other's build results recursively. For example, the Latex builder from Section 3 requires the *bbl* file generated by the Bibtex builder and the Bibtex builder requires the *aux* file generated by the Latex builder. Without special support for cycles, the Latex and Bibtex builders need to be merged into a single builder whose build procedure needs to manually take care to invoke Latex often enough and not to invoke Bibtex when it can be safely skipped.

In general, there is no one correct way of handling cycles and different builders require different cycle-handling strategies. Even though we did not show this in the definition of builders B in Figure 5, *pluto* allows builders to optionally provide custom cycle supports of the following form:

$$cycle\text{-}support : \overline{(B \times I)} \times \Omega \to (\overline{U})_\perp \times \Omega$$

Note that a cycle support has the same signature as a build system (Section 4.3). However, when a cycle support is called, it is guaranteed that the input builder/build-input pairs form a cycle if they were built normally. The only job of the cycle support is to break the cycle, that is, to implement a strategy for building the cycle such that there is a build unit for each builder in the cycle. We expect that most cycle-handling strategies are specific to the builders that occur in the cycle. Therefore, we allow a cycle support to reject building a cycle by returning $\perp$. If no builder in a cycle provides a handler capable of handling the cycle, *pluto* aborts the build with an appropriate error message.

To be sound, a cycle support must satisfy the same soundness criteria as a build system. That is, the result must form a well-formed dependency graph, each build unit was built with the requested builder and input, and each build unit is totally consistent.

5. The *pluto* Incremental Build Algorithm

In this section, we present *pluto*'s incremental build algorithm and prove it sound and optimal. Figure 6 and Figure 7 show the build algorithm without cycle support. We discuss how we support cycles in Section 5.2, but cycles are not included in the following formalization of the algorithm. The build algorithm of *pluto* consists of four functions:

- Function `build` is the entry point of the build system and satisfies the soundness criteria from Section 4.3.
- Function `require` takes a build request and yields a corresponding build unit. This function implements the incrementalization of *pluto* and yields a previously built build unit if it is still consistent or triggers a rebuild otherwise. Like in our API, Builders are intended to call

```
1  var P_consistent, var P_req, var Gen
2  function build(⟨(b_1, i_1) ··· (b_n, i_n)⟩, ω_0)
3      P_consistent, P_req := ∅
4      Gen := {p ↦ ⊥ | p ∈ P}
5      var ω := ω_0
6      for j from 1 to n do
7          val (u_j, ω') := require(b_j, i_j, ω)
8          ω := ω'
9      return (⟨u_1 ··· u_n⟩,ω)
10 function require(b, i, ω_0)
11     var ω := ω_0
12     val u := read_U(b.path(i), ω)
13     if u = ⊥ then
14         return execute(b, i, ω)
15     if b.path(i) ∈ P_consistent then
16         return (u, ω)
17     if u.builder ≠ b ∨ u.input ≠ i then
18         return execute(b, i, ω)
19     foreach gen p (stamp s f) ∈ u.gens do
20         if f(p, ω) ≠ s then
21             return execute(b, i, ω)
22     foreach r ∈ u.req do
23         if (freq p (stamp s f)) := r ∧ f(p, ω) ≠ s then
24             return execute(b, i, ω)
25         else if (breq b' i') := r then
26             (_, ω) := require(b', i', ω)
27     validate(u, b.path(i), ω)
28     P_consistent := P_consistent ∪ {b.path(i)}
29     return (u, ω)

30 function execute(b, i, ω_0)
31     val (u, ω) := b.build(i, ω_0)
32     val ω' := write_U(u, b.path(i), ω)
33     validate(u, b.path(i), ω')
33     P_consistent := P_consistent ∪ {b.path(i)}
34     return (u, ω')
```

Figure 6. Rebuild algorithm without cycle support.

function `require` in order to require the execution of another builder.

- Function `execute` runs a builder when `require` finds it is inconsistent.
- Function `validate` dynamically checks if the involved build units form a well-formed dependency graph.

We use three global variables that are initialized by function `build` at the beginning of each build-system run. Variable $P_{consistent}$ stores the set of consistent build units that have been checked or rebuilt during the current run. Thus, $P_{consistent}$ functions as a cache. Moreover, at all times, the build units in $P_{consistent}$ form a well-formed dependency graph. Variable P_{req} stores the set of files that have been required by build units in $P_{consistent}$. Variable Gen stores a mapping from a generated file to the path of the build unit that generated it. We use variables P_{req} and Gen in `validate` to check that dependencies are well-formed. After initializing the variables, function `build` simply iterates over the list of requested builds and calls `require` on each of them (Line 7). Note that we update the current file system ω during the loop.

```
1  function validate(u, p_u, ω)
2      if p_u ∈ P_consistent then
3          abort (⊥, ω)
4      foreach (gen p _) ∈ u.gens do
5          if Gen(p) ≠ ⊥ ∨ p ∈ P_req then
6              abort (⊥, ω)
7          else
8              Gen := {p ↦ p_u} ∪ Gen
9      for k from 1 to |u.reqs| do
10         if (freq p _) := u.reqs[k] then
11             P_req := P_req ∪ {p}
12             if Gen(p) ≠ ⊥ then
13                 val u_gen := read_U(Gen(p), ω)
                   val u_k := {builder := u.builder,
14                             input := u.input,
                               reqs := u.reqs[1 ... (k−1)],
                               gens := u.gens}
14                 if ¬ (u_k requires*_ω u_gen) then
15                     abort (⊥, ω)
```

Figure 7. Validation of dependency well-formedness.

Function `require` tries to reuse a previously built build unit u by reading it from the file system (Line 12). If no build unit is found, `require` triggers a rebuild using function `execute` (Line 14). If u exists and is known to be consistent, `require` simply returns it (Line 16). Otherwise, `require` performs a consistency check on u to determine whether a rebuild is necessary. In accordance with our definition of internal consistency in Section 4.2, a rebuild is necessary (i) if u was built by another builder or another input than the request one (Line 17), (ii) if any of the files generated by u is outdated (Line 19), or (iii) if any of the files required by u is outdated (Line 23).

However, function `require` not only checks for internal consistency, but also ensures total consistency. To this end, `require` recursively calls itself for any build required by u (Line 26), which may or may not lead to a rebuild of the required unit. After the recursive call returns, the required build is consistent and it is sound to use any of the files generated by it. Here, it is visible why a build must be required prior to requiring a file generated by that build: If the file was required first, the file requirement would already have been checked by the loop. Conversely, if the file is required after the build, the file requirement will

be subsequently checked by the loop. If no inconsistency was found, `require` validates the well-formedness of the dependency graph, caches the result, and returns the build unit (Line 27 ff.).

When called by `require`, function `execute` simply runs the requested builder, stores the build result, validates the well-formedness of the dependency graph, caches the result, and returns it.

Function `validate` (Figure 7) checks the well-formedness of the dependency graph induced by $P_{consistent}$. It does so incrementally by checking every build unit u added to the graph, such that all four well-formedness conditions from Section 4.1 hold. First, a graph is closed under *requires* because we call `validate` for every unit added to $P_{consistent}$. Then, `validate` checks that no previously added unit uses the same path as u (Line 2), and that no file generated by u overwrites a previously generated file (Line 4). Finally, `validate` checks that there are no hidden dependencies. That is, if path p is generated by u, then there is no previous requirement on p (Line 4), and if path p is required by u and p is generated by u_{gen}, then u transitively requires u_{gen} before it required p. To check the latter condition, `validate` constructs a build unit u_k that contains only those requirements that appear before the one to p. If `validate` finds a well-formedness violation, it aborts all pending builds and yields an error $\bot$.

5.1 Properties of *pluto*

Function `build` as defined above implements a sound and optimal build system as specified in Sections 4.3 and 4.4. Note that we presently only deal with non-cyclic builds. Proofs of all lemmas and theorems appear in Appendix A.

Soundness. To show soundness, we first observe that the global variable $P_{consistent}$ invariably stores a well-formed dependency graph. Moreover, P_{req} contains all files required by units in $P_{consistent}$, and Gen maps each file generated by a unit in $P_{consistent}$ to the path of that unit. Together, we call this the well-formedness invariant (details in Appendix A).

Lemma 5.1. *Given a call* $\texttt{require}(b, i, \omega_0) = (u, \omega)$*, if the well-formedness invariant holds for* ω_0*, then the well-formedness invariant also holds for* ω*.*

This holds because we use `validate` to explicitly check that adding a unit to $P_{consistent}$ retains well-formedness. Besides well-formedness, the units in $P_{consistent}$ are also totally consistent:

Lemma 5.2. *During a single run of* `build`*, whenever* $\texttt{path}(u)$ *is added to* $P_{consistent}$ *and the current file system is* ω*, then* u *is totally consistent in* ω*.*

This holds because (i) in a well-formed dependency graph, builders cannot invalidate each other, and (ii) we only add build units to $P_{consistent}$ after either ensuring their internal consistency or executing them (relying on Assumption 4.1).

From these lemmas, we can show that `build` is sound.

Theorem 5.3. *Function* `build` *satisfies the soundness criteria (S1), (S2), and (S3) from Section 4.3.*

Optimality. We first observe that due to the cache $P_{consistent}$, `build` executes each build request (b, i) at most once.

Lemma 5.4. *If* $\texttt{require}(b, i, \omega_0) = (u, \omega_1)$ *followed by* $\texttt{require}(b, i, \omega_2)$ *during a single run of* `build`*, then the second call yields a cached result in Line 16.*

Next, we show that `build` does not execute build requests (b, i) that are not (transitively) required by the result of `build`. Since all executed build requests add an entry to $P_{consistent}$, we show that $P_{consistent}$ only contains entries transitively required by the result of `build`.

Lemma 5.5. *If* $\texttt{build}(\overline{(b, i)}, \omega_0) = (\overline{u}, \omega)$*, then for all* $p \in P_{consistent}$ *with* $v = \mathsf{read}_U(p, \omega)$*,* u $requires^*_\omega$ v *for some* $u \in \overline{u}$*.*

Finally, if `build` executes a build request (b, i), then either there was no valid persisted build unit at $b.path(i)$ or the build unit had an outdated file requirement of a file p that was guaranteed to remain unchanged during the rest of the build.

Lemma 5.6. *Let* `build` *call* $\texttt{execute}(b, i, \omega_0) = (\_, \omega)$ *and* $p = b.path(i)$*. Then either*

1. $\mathsf{read}_U(p, \omega_0) = \bot$,
2. $\mathsf{read}_U(p, \omega_0) = u$ *such that* $u.builder \neq b$ *or* $u.input \neq i$,
3. $\mathsf{read}_U(p, \omega_0) = u$ *such that* $(\mathsf{freq}\ p\ s) \in u.reqs$ *outdated in* ω_0 *and* $Gen(p) = \bot$*, or*
4. $\mathsf{read}_U(p, \omega_0) = u$ *such that* $(\mathsf{freq}\ p\ s) \in u.reqs$ *outdated in* ω*,* $Gen(p) = p_v$*, and* $\mathsf{read}(p_v, \omega)$ *is totally consistent.*

With these lemmas together, we can show that the number of builders executed by *pluto* is minimal.

Theorem 5.7. *The number of builders executed by* `build` *is minimal.*

Lastly, we show that `build` executes the minimal number internal consistency checks (Line 19 ff.). For a call $build(\overline{(b, i)}, \omega_0) = (\overline{u}, \omega)$, the minimal number of consistency checks is

$$N = |\{v | u \in \overline{u}, u\ requires^*_\omega\ v, \mathsf{read}_U(path(v), \omega_0) \neq \bot\}|.$$

That is, the build system should try to reuse every build unit that is required in the result $\overline{u}$ for which existed a previous build in ω_0.

Theorem 5.8. *The number of consistency checks performed by* `build` *is* N*.*

5.2 Supporting Cycles

Our implementation supports cyclic dependencies as presented in Section 3.4. To support cycles in our rebuild algorithm, we use two stacks for detecting cycles and incrementally maintaining the result of a strongly-connected-component analysis on the growing dependency graph.

The first stack is used by function `execute` to detect cycles. Function `execute` adds a builder and its input before running it and removes them when the builder finishes. If a builder and its input are already on the stack when adding them, a cycle is found. The cycle involves all builders on the stack starting at the first occurrence of the cyclic builder.

All the builders in the cycle are part of a single strongly connected component in the dependency graph. But it might well be that other builders are in the same strongly connected component, too. For example, if builder A requires builder B, builder C requires B, and builder B requires both A and C (for simplicity, the inputs of the builders are left out). Then, A, B, and C form a strongly connected component, which also contains two cycles, namely A-B and B-C. The problem is that these two cycles mutually recursively require each other via B. Let us assume that the cycle A-B is detected first but the requirement on C was not found yet. During the run o the cycle handler for A-B, a requirement from B to C may be detected, leading to the execution of builder C (but not as part of the cycle handler). But then, the cycle B-C is detected and a new cycle handler is started. If this cycle handler was only run on B-C, the cycle A-B would be detected again, leading to an endless chain of mutually dependent cycle handlers. Our solution to this problem is to run the cycle handler on the full strongly connected component A-B-C, as soon as the second cycle is detected. It is not possible to run the cycle handler on A-B-C from the beginning because the requirement on C is not known yet.

The second stack is used by function `require` to determine the consistency of cycles. As shown in Figure 6, we determine consistency of a builder by transitively checking consistency for all required builders. But in a cycle, this transitive check includes the initial builder itself. We use the second stack to detect and break such requirement cycles by assuming the recursive build requirement to be consistent. Essentially, this just means that the cycle itself is not a source of inconsistency.

The support for cycles introduced some engineering overhead: the two stacks, the strongly-connected-component analysis, and the incremental detection of cycles in `execute` calls. But, as long as no cycles are detected, the rebuild algorithm implemented by us behaves exactly as the one formalized in this section.

5.3 Practical Considerations

We discuss a few practical considerations regarding the build algorithm of *pluto*. First, in our implementation, we hide a lot of the internal details of build units through our API. In particular, a builder in our API does not have to produce a build unit on its own, but simply register requirements and generated files. Second, in practice, builders frequently fail. In our model, builders can fail as long as they yield an internally consistent build unit nonetheless. In our implementation, a builder can simply throw an exception and the API will yield an appropriate build unit. A failing build will only be rerun if any of its requirements changes. Third, in contrast to the algorithm presented here, our implementation does not read a build unit from the file system all the time. Instead, we use an additional cache to save time. The source code of *pluto* is available online at

`https://github.com/pluto-build/pluto`.

6. Case Studies

We have used *pluto* in three different projects to support incremental building. First, we have migrated an existing Ant build script to *pluto*. We compare the two build scripts and report on their respective build times. Second, we have used *pluto* to implement Java-style separate compilation [1] for two existing languages. These case studies confirm *pluto*'s broad applicability and support for incremental building. The first case study moreover demonstrates that incremental building can improve build performance significantly.

6.1 From Ant to *pluto*: The Spoofax Builder

We manually reimplemented an existing Ant build script developed by others for the Spoofax language workbench [9]. Spoofax supports developers of external domain-specific languages (DSLs) by providing a set of language-definition languages. Spoofax provides languages for defining a DSL's syntax, name binding, type system, code generation, and editor services such as syntax coloring or code completion. Spoofax uses a fairly complex Ant build script[4] to compile all language-definition artifacts into an Eclipse plug-in. The Ant build script supports a rather limited form of incrementality based on the `uptodate` target. We reimplemented this build script using *pluto*.[5] For testing and for comparing build times, we used one of Spoofax's own DSLs.[6]

Figure 8 compares the original Ant build script to our *pluto* build script. For most Ant builders, we simply provide a corresponding implementation as a *pluto* builder. However, the Ant script has more builders (Ant targets) than *pluto* because we only implemented a *pluto* builder when the incremental pay-off seemed worthwhile. In particular, we implemented 12 computationally inexpensive Ant builders as regular Java methods that get called from actual `build` methods. Conversely, for one computationally expensive Ant builder, we added an additional builder for its subtasks in order to achieve better incrementalization. The implementation of our *pluto* build script is 58% longer than the Ant script. We believe this is admissible, given that Ant provides a domain-specific notation whereas we rely on regular Java code. The development of a concise DSL on top of *pluto* is outside the scope of this paper.

[4] `https://github.com/metaborg/sdf/blob/master/org.strategoxt.imp.editors.template/build.main.xml`

[5] `https://github.com/pluto-build/build-spoofax`

[6] `https://github.com/metaborg/sdf/tree/master/org.strategoxt.imp.editors.template`

	Ant	*pluto*
Builders	34	23
Lines of code	1168	1852
Builder dependencies	37	60
Hidden dependencies	6	0
Injected dependencies	0	5
Generated files	n/a	243
Required files (user-supplied)	n/a	108
Required files (generated)	n/a	208
Last-modified stamp	always	271
File-hash stamp	0	277
File-exists stamp	0	8
Custom stamp	0	3

Figure 8. Comparison of Ant and *pluto* build scripts.

Regarding builder dependencies, the Ant and *pluto* build scripts vary because of the removal and addition of builders as well as the discovery of previously hidden dependencies. For example, the *pluto* build script contains 34 additional dependencies due to dependencies on the subtask builder we added. Using our dynamic analysis, we discovered 6 dependencies that were hidden in the Ant script. We added builder dependencies in the *pluto* script accordingly. Finally, the *pluto* script uses generic builders for Stratego [17] compilation, for Java compilation, and for Java packaging in a *jar* file. Since these builders deal with generated files, the *pluto* build script injects a total of 5 dependencies.

Figure 8 also provides some statistics about required and provided files for the *pluto* build script. In total, a build required 108 user-supplied source files and generated 243 files, from which 208 were subsequently required by other builders. Our build script uses a last-modified stamp for all generated files as well as for 28 required files (larger files and binaries typically). We use a file hash for almost all other requirements. As exception, in 8 cases, a builder was only interested in whether a file exists or not. Finally, we implemented 3 custom stamper functions to precisely describe the requirements of 3 builders. These 3 builders depend on the grammar of the user-defined DSL, but each of the builders depends on a different aspect of the grammar: The first builder generates a pretty printer and ignores the names of nonterminals, the second builder generates a parenthesis optimizer and ignores everything except precedence declarations, and the third builder generates AST declarations and ignores all lexical syntax. Since the execution of these builders is quite costly (0.4–1.2 seconds each), being able to skip their execution when the grammar changed in irrelevant ways can save important time.

Incremental performance. To demonstrate the advantage of incremental building, we measured the build times of the *pluto* build script after various changes to the DSL definition. Since the Ant build script supports only limited incrementality, we did not compare incremental build times of the two systems but only within *pluto*. We repeated all measurements five times, discarded the first and second measurements, and report the average wall-clock time of the remaining three measurements. We report build times in seconds together with the speedup relative to a clean *pluto* build in Figure 9. The measurements were conducted on a dual core 2.8 GHz MacBook Pro with 8GB memory running Eclipse 4.4.2 with a maximum heap size of 512MB.

Code modification	Build time	Inc. speedup
Clean build	31.3 s	1
No change	0.027 s	1160
Grammar	16.0 s	1.9
Name analysis	5.7 s	5.5
Type system	4.7 s	6.7
Code generator	4.8 s	6.5
Editor service	1.2 s	26.1

Figure 9. Incremental Spoofax build times with *pluto*.

We first measured the build time of a clean *pluto* build when all builders have to be executed. The clean build time of 31.3 seconds serves as a reference value for the incremental rebuilds. For comparison, we also measured the clean build time of the original Ant script (not shown in the table), which amounts to 56.7 seconds and is slower than a clean *pluto* build by a factor of 0.55. While we have not investigated this slowdown in detail, the most likely explanation seems to be that the *pluto* build script runs in a single VM whereas each invocation of Ant instantiates a fresh VM. Using a single VM reduces the (accumulated) startup time and enables JIT optimizations across build-script runs.

As shown in Figure 9, we measured *pluto* build times after various changes of the DSL definition. When nothing is changed, a rebuild is almost instantaneous and only takes 27 milliseconds, which yields a dramatic speedup. A change to the grammar of the DSL definition entails quite some rebuilding of builders that depend on the grammar, such as the parse table, pretty-printer, abstract-syntax declarations, default structural editor services, and more. In total this amounts to 16.0 seconds and a speedup of only 1.9. Other changes are less invasive. A change to the name analysis, type system, or code generation leads to modified Stratego code, which the build script subsequently recompiles, yielding speedups between 5.5 and 6.7 compared to a clean build. Finally, we measured the build time for changing a DSL's editor services by changing the color for highlighting keywords. While the definition of editor services typically depends on many other DSL definitions like the grammar, the editor services are part of the final output of the Spoofax builder and no other builders depend on them. For this reason, a change to a DSL's editor services entails little rebuilding and yields a high speedup of 26.1.

In summary, we can say that *pluto* provides substantial speedups for rebuilding a Spoofax DSL definition after a change. As expected, the actual speedup depends on how many builders depend on the changed artifact transitively.

6.2 Implementing Compilers with *pluto*

A build system is not the only software artifact that generates files and manages dependencies. Another typical example are compilers. A compiler takes a set of source files as input and compiles them into binaries of some form. When any of the source files changes, the compiler needs to rebuild all binaries that depend on the changed source file. *pluto* can be readily used to implement such compilers that are incremental at the file level.

To implement a compiler with *pluto*, the compiler must be defined as a builder using our API from Section 3.1. That is, the compiler extends class Builder and provides a method build as entry point to the compilation. The next step is to organize builder invocations such that each builder only compiles a single source file. When a source file depends on the compilation result of other source files (e.g., due to an import statement), the builder issues a build request to itself with the other source file as input. Finally, the builder registers a file dependency on binaries that it needs during compilation. A compiler like this spans a dependency graph for which our rebuild algorithm automatically provides incremental recompilation. Furthermore, the builder can use *pluto*'s customizable stampers to express precise dependencies. For example, a Java compiler could use a custom stamper to only depend on the class header, field types, and method signatures within a class file, but not on the bodies of methods.

Following this method, we have used *pluto* to implement file-level incremental compilation for two programming languages, Stratego [17] and SugarJ [3, 5]. Previously, Stratego did not feature separate compilation and performed a global compilation with all source files at once. SugarJ did feature file-level incremental compilation before, implementing its own dependency tracking and suboptimal rebuild algorithm. Based on *pluto*, we were able to eliminate this code, relying instead on the more general, sound, and optimal dependency tracking and rebuilding of *pluto*. The code for our builders of Stratego[7] and SugarJ[8] is available online.

The Stratego and SugarJ case studies show that *pluto* is applicable to different application scenarios, including the development of file-level incremental compilers. In particular, it is possible to reuse the dependency tracking and rebuild algorithm of *pluto*, which typically is a complicated subsystem of a compiler that now can be eliminated.

[7] https://github.com/lichtemo/strategoxt/tree/separate-compilation-bootstrap

[8] https://github.com/sugar-lang/compiler

7. Related Work

pluto follows in the footsteps of a large number of existing build systems and a full survey of them is not in scope of this paper. For our discussion of related work, we selected representatives for different styles of dependency management and incremental rebuilding.

First, we would like to point out that, to the best of our knowledge, *pluto* is the only build system that supports incremental building of Latex documents with exact dependencies. Even Latex-specific build systems like *latexmk*[9] use unsound heuristics to detect dependencies.

We already discussed Make, Shake, and Ant in Section 2. To summarize, Make [14, 16] does neither support dynamic discovery of file requirements, custom file stamps, metadependencies, nor cycles. Therefore, in order to achieve sound incremental rebuilding with Make, build-script developers have to overapproximate the set of file dependencies, for example, by requiring all Java files in a source directory even though only a few of them are needed. Shake [13] supports the dynamic discovery of required files and inspired *pluto*. We improve over Shake by supporting custom file stamps, metadependencies, and cycles. In contrast, Shake uses the last-modified time, does not detect build-script changes, and aborts execution when detecting a cycle. Ant makes the skipping of build rules programmable. In order to use this mechanism in a sound way, build-script developers essentially have to implement their own dependency-management system. Indeed, it might be possible to implement the concepts we developed for *pluto* on top of Ant as macros.

The majority of existing build systems model dependencies identical to Make. Examples include MSBuild, Ninja, SCons, Gradle, and CloudMake. The features they add on top of Make are unrelated to dependency management and incremental rebuilding, such as the syntax of build scripts, the execution environment, and parallelization.

Gradle does not support dynamically discovered dependencies. Dependencies are declared as part of a task using annotations @InputDirectory and @InputFile. In addition, Gradle allows dependencies to be added *between* builds, but not during a build. The lack of dynamic dependencies also becomes apparent when inspecting Gradle's build lifecycle.[10] Gradle first identifies all outdated tasks and then executes all of them. With dynamic dependencies this would be wrong, because the execution of one task could produce a new task dependency or drop an old dependency. Gradle also supports Ant-like up-to-date checks through method TaskOutputs.upToDateWhen.

CloudMake [2] has Make-like, fixed dependencies but uses a functional programming language for describing builds. In contrast to most other Make derivatives, CloudMake features a formal specification and soundness proof, which shows that a very simple caching strategy yields sound incre-

[9] http://www.ctan.org/pkg/latexmk/

[10] http://gradle.org/docs/0.8/userguide/build_lifecycle.html

mental building. Our build system features more complex and dynamic dependencies than CloudMake, requiring a more sophisticated rebuild algorithm and a more involved soundness proof. Metamorphosis [6] is a tool for automatically migrating existing build scripts to CloudMake. Metamorphosis provides a dynamic analysis that traces opened file handles at the operating-system level in order to determine a build script's required and provided files. From this, Metamorphosis infers approximate Make-like file dependencies. Interestingly, beyond migration, the dynamic analysis can also be used to validate the correctness of the dependency declarations of a build rule, that is, to check that the dynamically required and provided files are a subset of the declared file dependencies. It would be interesting to combine this dynamic analysis with *pluto*, where we could simply run the analysis with every build and register exact dependencies dynamically.

Tup [15] features Make-like build rules that declare file dependencies statically. However, the static dependencies do not need to mention non-generated required files. Tup monitors the file system in the working directory to infer the files read and written during a build-script run. Tup compares the actually required and provided files to the statically declared ones and, similar to *pluto*, raises an error if there are any hidden dependencies. By monitoring the file system, Tup also detects changed files (including changed build scripts) and feeds this information into its rebuild algorithm to identify inconsistent build results quickly. However, in contrast to *pluto*, Tup requires static dependencies and does not support custom file stamps or cycles.

Vesta [7] is an integrated version-control and build system that tracks file changes and supports incremental building. Vesta works on immutable snapshots of source-code repositories. Thus, a Vesta build script is a purely functional program operating on immutable data. Vesta's build algorithm evaluates the build scripts and uses a dynamic analysis for fine-grained dependency analysis [8]. These dependencies are not restricted to files but record requirements on function arguments as well. As such, Vesta implements incremental evaluation for a purely functional build-script language on top of an immutable file system. Unlike Vesta, *pluto* places no constraints on its environment and supports regular file systems by checking file consistency explicitly and regular programming languages by dynamically enforcing soundness. While Vesta infers precise functional dependencies of called build-script functions, *pluto* builders can declare fine-grained dependencies using file stamps.

SBT[11] is a build system for Scala and Java source files. SBT aims at capturing precise dependencies between source files in order to support incremental recompilation. To this end, SBT intercepts dependency information generated by the Scala compiler and constructs a dependency graph that only contains source files but no generated files. When a source file changes, SBT computes the transitive closure of all source files that are potentially affected by the change and recompiles all of them at once. In contrast, *pluto* also tracks requirements on generated files and triggers rebuilds incrementally, because a rebuild may or may not lead to subsequent inconsistencies.

Bazel[12] is a new build system developed by Google that advertises incrementality and correctness. While Bazel did not influence the design of *pluto*, Bazel's dependency graphs turned out to have the same form as the dependency graphs of *pluto*: Builders require other builders, and builders require and provide files. In contrast to *pluto*, Bazel does not support dynamically discovered dependencies. Instead, Bazel operates in three phases, where it first loads the build configurations, then constructs the complete dependency graph, and finally triggers builders as required. *pluto* interleaves dependency analysis and builder execution to allow the latter to influence the former.

8. Conclusions and Future Work

We presented *pluto*, a build system with sound and optimal incremental building. *pluto* ensures soundness through a dynamic analysis that enforces invariants on the dependency graph. To support good incrementality, *pluto* provides mechanisms for fine-grained dependency description, in particular, dynamic dependency discovery, custom file stamps, metadependencies, and cycles. We have verified *pluto*'s soundness and optimality and demonstrated its practical applicability and incrementality benefits.

In future work, we want to support parallel builds on top of incremental building, which should be easy since we already model dependencies precisely. Moreover, we want to investigate support for the automated inference of file dependencies and file stamps. Finally, we want to explore tools for migrating existing build scripts to *pluto*, which should be easy because dependencies in *pluto* subsume dependencies in other build systems.

Acknowledgments

We thank Mira Mezini, Klaus Ostermann, and the anonymous reviewers for helpful feedback on this work. This work was supported in part by Oracle Labs and by the European Research Council, grant No. 321217.

References

[1] D. Ancona, F. Damiani, S. Drossopoulou, and E. Zucca. Polymorphic bytecode: Compositional compilation for java-like languages. In *Proceedings of Symposium on Principles of Programming Languages (POPL)*, pages 26–37. ACM, 2005.

[2] M. Christakis, K. R. M. Leino, and W. Schulte. Formalizing and verifying a modern build language. In *Proceedings of Symposium on Formal Methods*, volume 8442 of *LNCS*, pages 643–657. Springer, 2014.

[11] `http://www.scala-sbt.org/`

[12] `http://bazel.io/`

[3] S. Erdweg. *Extensible Languages for Flexible and Principled Domain Abstraction*. PhD thesis, Philipps-Universiät Marburg, 2013.

[4] S. Erdweg and K. Ostermann. Featherweight TeX and parser correctness. In *Proceedings of Conference on Software Language Engineering (SLE)*, volume 6563 of *LNCS*, pages 397–416. Springer, 2010.

[5] S. Erdweg and F. Rieger. A framework for extensible languages. In *Proceedings of Conference on Generative Programming and Component Engineering (GPCE)*, pages 3–12. ACM, 2013.

[6] M. Gligoric, W. Schulte, C. Prasad, D. van Velzen, I. Narasamdya, and B. Livshits. Automated migration of build scripts using dynamic analysis and search-based refactoring. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 599–616. ACM, 2014.

[7] A. Heydon, R. Levin, T. Mann, and Y. Yu. The Vesta approach to software configuration management. Technical Report 168, Compaq SRC, 2001.

[8] A. Heydon, R. Levin, and Y. Yu. Caching function calls using precise dependencies. In *Proceedings of Conference on Programming Language Design and Implementation (PLDI)*, pages 311–320. ACM, 2000.

[9] L. C. L. Kats and E. Visser. The Spoofax language workbench: Rules for declarative specification of languages and IDEs. In *Proceedings of Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 444–463. ACM, 2010.

[10] A. Kuhn, G. C. Murphy, and C. A. Thompson. An exploratory study of forces and frictions affecting large-scale model-driven development. In *Proceedings of Conference on Model Driven Engineering Languages and Systems (MoDELS)*, volume 7590 of *LNCS*, pages 352–367. Springer, 2012.

[11] E. T. Kumfert G. Software in the DOE: The hidden overhead of "The Build". Technical report, Lawrence Livermore National Laboratory, 2002.

[12] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of ANT build systems. In *Proceedings of Conference on Mining Software Repositories (MSR)*, pages 42–51, 2010.

[13] N. Mitchell. Shake before building: Replacing make with Haskell. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 55–66. ACM, 2012.

[14] A. Neagu. What is wrong with Make? Available at `http://freecode.com/articles/what-is-wrong-with-make`, 2005.

[15] M. Shal. Build system rules and algorithms. Available at `http://gittup.org/tup/build_system_rules_and_algorithms.pdf`, 2009.

[16] R. M. Stallman, R. McGrath, and P. D. Smith. *GNU Make Manual*. Free Software Foundation, 2014.

[17] E. Visser, Z.-E.-A. Benaissa, and A. P. Tolmach. Building program optimizers with rewriting strategies. In *Proceedings of International Conference on Functional Programming (ICFP)*, pages 13–26. ACM, 1998.

A. Proofs and Auxiliary Lemmas for Section 5

Lemma A.1. *If* `require`$(b, i, \omega_0) = (u, \omega)$*, then* $\mathsf{path}(u) \in P_{consistent}$ *and* $\mathsf{read}_U(\mathsf{path}(u), \omega) = u$ *after the call.*

Proof. `require` returns from one of the following lines:

Lines 14, 18 or 24: The function terminates with a call to `execute`, which puts $\mathsf{path}(u)$ in $P_{consistent}$ and writes it to ω.

Line 16: Since $\mathsf{path}(u) = b.path(i)$ by Assumption 4.1, $\mathsf{path}(u)$ already in $P_{consistent}$ and ω_0.

Line 29 Line 28 puts $\mathsf{path}(u)$ in $P_{consistent}$ and $\mathsf{path}(u)$ already in ω_0. □

Lemma A.2. *During a single run of* `build`*, whenever* $\mathsf{path}(u)$ *is added to* $P_{consistent}$ *and* $(\mathsf{breq}\ b\ i) \in u.reqs$*, then* $b.path(i) \in P_{consistent}$.

Proof. Either Line 28 or Line 33 was executed to add u to $P_{consistent}$.

Line 28: Since $(\mathsf{breq}\ b\ i) \in u.reqs$, `require`$(b, i, \omega)$ was executed in Line 26.

Line 33: Since $(\mathsf{breq}\ b\ i) \in u.reqs$, $b.build(i)$ called `require`(b, i, ω).

In both cases, $b.path(i) \in P_{consistent}$ by Lemma A.1. □

Lemma A.3. *During a single run of* `build`*, whenever* $\mathsf{path}(u)$ *is added to* $P_{consistent}$ *and the current file system is* ω*, then* $\mathsf{path}(v) \in P_{consistent}$ *for all* u $requires^*_\omega$ v.

Proof. By induction on the build-requirements structure of u, using Lemma A.2. □

Lemma A.4. *Let* u *be a build unit. Then its summary file* $\mathsf{path}(u)$ *is written at most once during a* `build` *run.*

Proof. A summary file is only written in line 32. Then line 33 is executed too and thus $\mathsf{path}(u) \in P_{consistent}$. No unit is ever removed from this set. Thus, any subsequent call to `require` with $b.path(i) = \mathsf{path}(u)$ terminates in Line 16. □

Lemma A.5. *If* `require`$(b, i, \omega_0) = (u, \omega)$*, then* $u.builder = b$ *and* $u.input = i$.

Proof. By Lemma A.1, $u \in P_{consistent}$. Thus, either Line 28 was executed and $u.builder = b$ and $u.input = i$ holds by Line 17, or Line 33 was executed and $u.builder = b$ and $u.input = i$ holds by Assumption 4.1. □

Definition A.6 (Well-formedness invariant)**.** Let $reqs_f(u) = \{p | (\mathsf{freq}\ p\ \_) \in u.reqs\}$ be the function, which extracts all paths to required files for a $u \in U$. Then we call the following conditions the well-formedness invariant:

a) The set of units located at the paths $P_{consistent}$ is a ω-well-formed dependency graph,

b) $P_{req} = \bigcup_{p \in P_{consistent}} reqs_f(\mathsf{read}_U(p, \omega))$ and

c) $Gen(p') = \begin{cases} u & \text{if } \exists p \in P_{consistent}.p' \in \mathsf{read}_U(p,\omega).gens \\ \bot & \text{otherwise} \end{cases}$

Lemma A.7. *Let $u \in U$ and $\mathsf{path}(u) \notin P_{consistent}$. If the well-formedness invariant holds for some ω, and for all $v \neq u$ with u $requires^*_\omega$ v we have $\mathsf{path}(v) \in P_{consistent}$, then the well-formedness invariant holds for ω after a non-aborting execution of* `validate`*$(u, \mathsf{path}(u), \omega)$ and $P_{consistent} := P_{consistent} \cup \{\mathsf{path}(u)\}$.*

Proof. We only add u to $P_{consistent}$. Thus, it suffices to show that u does not introduce a violation of the invariant. First, the execution of `validate` adds entries to P_{req} and Gen in accordance with the invariant, satisfying conditions b) and c). Condition a) holds because u retains the well-formedness of the dependency graph induced by $P_{consistent}$:

1. $P_{consistent}$ is still closed under $requires_\omega$ because we assumed that the path of all units required by u is already in $P_{consistent}$.
2. `validate` checked that $\mathsf{path}(u)$ does not overlap with existing paths in $P_{consistent}$.
3. `validate` checked that $u.gens$ does not overlap with previously generated files, all of which are in Gen.
4. `validate` checked that no $p \in u.gens$ was previously required and for all files p required by u, if p was previously generated by v, then u requires v through a build requirement that occurs before the file requirement of p.

□

Proof for Lemma 5.1. By induction on the recursive structure of `require`.

Base Case: Let `require`$(b, i, \omega_0) = (u, \omega)$ not make any recursive call, neither directly or indirectly through `execute`. If $\mathsf{path}(u) \in P_{consistent}$ before the call, $P_{consistent}$, P_{req}, Gen are unchanged and $\omega = \omega_0$. Thus, the invariant still holds. Otherwise, `require` returns by an `execute` call or from line 29. In both cases, since there is no recursive call to `require`, $P_{consistent}$, Gen and P_{req} are unchanged before line 33 and 27, respectively. Moreover, u does not have any build requirements. Therefore, we can apply A.7 and the invariant holds in both cases.

Inductive Case: Let `require`$(b, i, \omega_0) = (u, \omega)$. If $\mathsf{path}(u) \in P_{consistent}$ before the call, $P_{consistent}$, P_{req}, Gen are unchanged and $\omega = \omega_0$. Thus, the invariant still holds. Otherwise, `require` may make a series of recursive calls, all of which retain the well-formedness invariant due to the induction hypothesis. At last, `require` returns by an `execute` call or from line 29. In both cases, by Lemma A.3, all units transitively required by u are in $P_{consistent}$. Therefore, we can apply A.7 and the invariant holds in both cases.

□

Lemma A.8. *During a single run of* `build`*, whenever $\mathsf{path}(u)$ is added to $P_{consistent}$ and the current file system is ω, then u is internally consistent in ω. Moreover, all other units v with $\mathsf{path}(v) \in P_{consistent}$ remain internally consistent.*

Proof. Either Line 28 or Line 33 was executed to add u to $P_{consistent}$.

Line 28: u is internally consistent because all generated and required files are up-to-date and, by Lemma A.5, the required build units have fields $builder$ and $input$ correctly set.

Line 33: u was generated by the build function and is internally consistent due to Assumption 4.1.

In each case, function `validate` was executed to ensure that u does not invalidate any previously added units v with $\mathsf{path}(v) \in P_{consistent}$. □

Proof of Lemma 5.2. By induction on the build-requirements structure of u, using Lemmas A.4 and A.8, u and all v with u $requires^*_\omega$ v are internally consistent. So u is totally consistent. □

Proof of Theorem 5.3. By induction on the number of elements in the sequence $\overline{(b, i)}$. The base case is trivial. For the step case, assume we already have results $\langle u_1, \ldots, u_n$ and build request (b_{n+1}, i_{n+1}) next.

(S1) By the induction hypothesis, the dependency graph induced by $\langle u_1, \ldots, u_n\rangle$ is ω-well-formed. By Lemma 5.1, we get that dependency graph induced by $\langle u_1, \ldots, u_{n+1}\rangle$ is ω'-well-formed as required.

(S2) By the induction hypothesis, all results $\langle u_1, \ldots, u_n\rangle$ have fields $builder$ and $input$ correctly set. By Lemma A.5, result u_{n+1} also the right fields.

(S3) By induction hypothesis, all previous results are totally consistent. By Lemma A.1, $\mathsf{path}(u_{n+1}) \in P_{consistent}$ after the call `require`$(b_{n+1}, i_{n+1}, \omega) = (u_{n+1}, \omega')$. By Lemma 5.2, u_{n+1} is totally consistent in ω', as are all other $\langle u_1, \ldots, u_n\rangle$.

□

Proof of Lemma 5.4. By Lemma A.1, $u \in P_{consistent}$ after `require`$(b, i, \omega_0) = (u, \omega_1)$ and $\mathsf{read}_U(\mathsf{path}(u), \omega_1) = u$. Thus, any subsequent call `require`(b, i, ω_2) will return (u, ω_2). □

Lemma A.9. *Let $P_0 := P_{consistent}$ before and $P_1 := P_{consistent}$ after a call* `require`*$(b, i, \omega_0) = (u, \omega_1)$. Then, for every $p \in P_1 \setminus P_0$ with $v = \mathsf{read}_U(p, \omega_1)$, u $requires^*_{\omega_1}$ v.*

Proof. By induction on the build-requirements structure of u.

Base Case: If u does not have any build requirements, than $P_1 \setminus P_0 \subseteq \{\mathsf{path}(u)\}$, which is reflexively required by u.

Inductive Case: Let u have build requirements $\langle r_1, \ldots, r_k \rangle$ that produce units $u_1, \ldots, u_k$ and add paths $Q_1, \ldots, Q_k$ to P_0. By induction hypothesis, each u_j requires all units referenced in Q_j. Furthermore, u requires all $u_1, \ldots, u_k$, and thus transitively all units referenced in $Q_1, \ldots, Q_k$. We have $P_1 \setminus P_0 \subseteq \bigcup Q_j \cup \{\mathsf{path}(u)\}$, all of which are required by u. □

Proof of Lemma 5.5. By structural induction on the sequence $\overline{(b, i)}$, using Lemma A.9. □

Proof of Lemma 5.6. `execute` is only called by `require`. `require` calls `execute` in the following lines:

Line 14: Here it is that, because $\omega = \omega_0$, $\mathsf{read}_U(b.path(i), \omega) = \bot$, so case 1. of the lemma.

Line 18: Then $u = \mathsf{read}_U(b.path(i), \omega)$ is defined but because of the if condition $u.builder \neq b$ or $u.input \neq i$, thus case 2. of the lemma.

Line 24: This call is made because an outdated file requirement (freq p _) is detected. Then either $Gen(p) = \bot$ or $Gen(p) = p_v$. Because of (S1) from Theorem 5.3, if p was generated by a unit v located at at p_v, then $Gen(p) = p_v \neq \bot$. If $Gen(p) = \bot$ case 3. of the lemma applies, because the file at p did not changed during `build`, because it is not generated, so outdated in ω_0. Otherwise by (S1) again, there was a build dependency to v before the file dependency. From (S3) and Lemma 5.2 follows, that v is totally consistent. And by Lemma A.4 and A.1, $\mathsf{read}_U(p_v, \omega) = v$ is totally consistent. Because p has been generated, it will not be overwritten, thus is outdated in ω. So the 4. case of the lemma applies.

This covers all calls to `execute`. □

Proof of Theorem 5.7. Lemma 5.4 shows, that a build unit is at most build once during a `build` call. Then Lemma 5.5 shows, that every unit in $P_{consistent}$ is in the dependency graph DG induced by $\bar{u}$. Lemma A.1 shows that as a result of each `require` call a path is put in $P_{consistent}$. Hence, only build units are required, which are in DG. This means, that no builders for units are executed, which are not used in the end and that no builder is executed twice for the same input. Lemma 5.6 shows finally, that a builder is only executed, if no existing build unit is found or the existing build unit u must be internally inconsistent. So the builder needs to be executed to make u totally consistent. □

Proof of Theorem 5.8. From 5.5 we know that every call to `require` will result in a build unit $u \in DG$. Because of 5.4 the number of `require` calls, which does not terminate in line 16, is $|DG|$. Only if `require(`b, i, ω_0`)` returns from line 14, $\mathsf{read}_U(b.path(i), \omega_0) = \bot$. Let the number of these calls be M. Then the number of consistency checks N' is

$$\begin{aligned} N' &= |DG| - M \\ &= |\{v \in DG\}| - |\{v \in DG | \mathsf{read}_U(\mathsf{path}(v), \omega_0) = \bot\}| \\ &= |\{v \in DG | \mathsf{read}_U(\mathsf{path}(v), \omega_0) \neq \bot\}| \\ &= N \end{aligned}$$ □