Scalable Incremental Building with Dynamic Task Dependencies

Konat, Gabriël; Erdweg, Sebastian; Visser, Eelco

DOI
10.1145/3238147.3238196

Publication date
2018

Document Version
Final published version

Published in
ASE 2018

Scalable Incremental Building with Dynamic Task Dependencies

Gabriël Konat
Delft University of Technology
Delft, The Netherlands
g.d.p.konat@tudelft.nl

Sebastian Erdweg
Delft University of Technology
Delft, The Netherlands
s.t.erdweg@tudelft.nl

Eelco Visser
Delft University of Technology
Delft, The Netherlands
e.visser@tudelft.nl

ABSTRACT

Incremental build systems are essential for fast, reproducible software builds. Incremental build systems enable short feedback cycles when they capture dependencies precisely and selectively execute build tasks efficiently. A much overlooked feature of build systems is the expressiveness of the scripting language, which directly influences the maintainability of build scripts. In this paper, we present a new incremental build algorithm that allows build engineers to use a full-fledged programming language with explicit task invocation, value and file inspection facilities, and conditional and iterative language constructs. In contrast to prior work on incrementality for such programmable builds, our algorithm scales with the number of tasks affected by a change and is independent of the size of the software project being built. Specifically, our algorithm accepts a set of changed files, transitively detects and re-executes affected build tasks, but also accounts for new task dependencies discovered during building. We have evaluated the performance of our algorithm in a real-world case study and confirm its scalability.

CCS CONCEPTS

• **Software and its engineering → Software configuration management and version control systems**;

KEYWORDS

scalable, incremental, build, dynamic task dependency

ACM Reference Format:
Gabriël Konat, Sebastian Erdweg, and Eelco Visser. 2018. Scalable Incremental Building with Dynamic Task Dependencies. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238196

1 INTRODUCTION

Virtually every large software project employs a build system to resolve dependencies, compile source code, and package binaries. One great feature of build systems besides build automation is incrementality: After a change to a source or configuration file, only part of a build script needs re-execution while other parts can be reused from a previous run. Indeed, incremental build systems are a key enabler for short feedback cycles. The reliable and long-term maintainable usage of incremental build systems requires the following three properties:

- **Efficiency**: The most obvious requirement is that rebuilds must be efficient. That is, the amount of time required for a rebuild must be proportional to how many build tasks are affected by a change. Specifically, a small change affecting few tasks should only incur a short rebuild time.
- **Precision**: An incremental rebuild is only useful if it yields the exact same result as a clean build. To this end, incremental build systems must capture precise dependency information about file usage and task invocations. Make-like build systems do not offer means for capturing precise dependencies. Instead, over-approximation (`*.h`) leads to inefficiency because of considering too many files, and under-approximation (`mylib.h`) leads to incorrect rebuilds because of missing dependencies (e.g., `other.h`). Precise dependency information is required for efficient and correct rebuilds.
- **Expressiveness**: Like all software artifacts, build scripts grow during a project's lifetime [19] and require increasing maintenance [17]. Therefore, build scripts should be written in expressive languages, avoiding accidental complexity. That is, build scripting languages should not require build engineers to apply complicated design patterns (e.g., recursive [20] or generated Makefiles) for expressing common scenarios.

Current incremental build systems put a clear focus on efficiency and precision, but fall short in terms of expressiveness. In particular, in order to support incremental rebuilds, current systems impose a strict separation of configuration and build stages. All variability of the build process needs to be fixed in the configuration stage, whereas the build stage merely executes a pre-configured build plan. This model contradicts reality, where *how to build an artifact* depends on the execution of other build tasks. We have observed two sources of variability in building. First, based on the result of other tasks, *conditional building* selects one of multiple build tasks to process a certain input. Second, based on the result of other tasks, *iterative building* invokes build tasks multiple times on different inputs. In both cases, dependencies on task invocations only emerge during the build; build engineers cannot describe these *dynamic dependencies* in the configuration phase. We illustrate a concrete example in Section 2.

A solution to the expressiveness problem is to provide build engineers with a full-fledged programming language. In such a system, build tasks are procedures that can invoke other build tasks in their body. Build tasks can inspect the output of invoked tasks and use that to conditionally and iteratively invoke further tasks. The problem of such a programmable build system is that it is

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09.
https://doi.org/10.1145/3238147.3238196

difficult to achieve incrementality. We are only aware of a single build system that is both programmable and incremental: Pluto [3]. Unfortunately, the incremental build algorithm of Pluto has an important limitation: To check which tasks need re-execution, Pluto needs to traverse the entire dependency graph of the previous build and has to touch every file that was read or written in the previous build. This contradicts our first requirement, *efficiency*, because the rebuild time of Pluto depends on the size of the software project more than it depends on the size of the change. In particular, even when no file was changed, Pluto's algorithm requires seconds to determine that indeed no task requires re-execution. We illustrate Pluto's algorithm using an example in Section 2.

In this paper, we design, implement, and evaluate a new incremental build algorithm for build systems with dynamic task dependencies. While Pluto's algorithm only takes the old dependency graph as input and traverses it top-down, our algorithm also takes a set of changed files and primarily traverses the dependency graph bottom-up. We can collect changed files, for example, from IDEs that manage their workspace or by using a file system watchdog. Our algorithm uses the changed files to drive rebuilding of tasks, only loading and executing those tasks that are (transitively) affected by a change. However, due to dynamic task dependencies, the dependency graph can change from one build to the next one. Our build algorithm accounts for newly discovered and deleted task dependencies by mixing bottom-up and top-down traversals.

Our new incremental build algorithm provides significant performance improvements when changes are small. We have conducted a real-world case study on the Spoofax language workbench, a tool built for developing domain-specific languages (DSLs). The build script of Spoofax processes DSL specification files and generates interpreters, compilers, and IDE plug-ins for them. We found that our algorithm successfully eliminates the overhead of large dependency graphs and provides efficient rebuilding that is proportional to the change size.

In summary, we make the following contributions. We review programmatic build scripts, incremental building with Pluto, and why this does not scale (Section 2). We describe our key idea of bottom-up incremental building, and what is needed to make it work (Section 3) We present our hybrid incremental build algorithm that mixes bottom-up and top-down building (Section 4), and briefly discuss its implementation (Section 5). We evaluate the performance of the hybrid algorithm against Pluto's algorithm with a case study on the Spoofax language workbench (Section 6).

2 BACKGROUND AND PROBLEM STATEMENT

Most build systems provide a *declarative* scripting language. Declarative languages are great as they let developers focus on *what* to compute rather than *how* to compute it. However, we argue that declarativity is misdirected when it comes to describing sophisticated build processes that involve conditional and iterative task application.

For example, consider the build script in Figure 1. We wrote this build script in the PIE build script language [16], which mostly provides standard programming language concepts. That is, the build script performs iterative building by defining and calling functions

```
func main() -> string {
  val config = parseYaml(./config.yaml)
  val src = config.srcDir
  if (config.checkStyle) {
    val styleOk = checkStyle(src)
    if (config.failOnStyle && !styleOk)
      return "style error"
  }
  val userTests = ./test/**
  val genTests = genTests(src, ./test-gen)
  var failed = 0
  for (test <- userTests ++ genTests) {
    val testOk = runTest(test)
    if (!testOk) failed += 1
  }
  return "Failed tests: " + failed
}
func parseYaml(p: path) -> Config {...}
func checkStyle(src: path) -> bool {...}
func genTests(src: path, trg: path) -> path* {...}
func runTest(test: path) -> int {...}
```

Figure 1: Build script that invokes tasks conditionally and iteratively at build time.

(tasks) like `main` and `parseYaml`, stores results of tasks in local variables such as `config` and `src` which can be immediately used by subsequent tasks, and involves conditional building with control structures like `if` and `for`. By and large, our build script is a normal program that happens to handle file paths and invoke external processes to generate and run tests. But how can we execute such a programmatic build script incrementally?

Most build systems require declarative specifications of build tasks for this reason: to support efficient incremental rebuilds. However, Erdweg et al. demonstrated that it is also possible to incrementally execute programmatic build scripts, with *Pluto* [3], a build system that incrementally executes build scripts written in Java. The PIE language we used in our example is an alternative front-end to Pluto [16].

The build algorithm of Pluto constructs a dependency graph of a build while the build script runs. For example, consider the dependency graph of our example script in Figure 2. The dependency graph contains a node for each invoked task and for each read or written file. Edges between nodes encode dependencies. A task depends on the tasks it invokes and on the files it reads or writes. Moreover, when a task reads a file that was generated by another task, the reading task depends on the generating task such that the generating task is executed first. While not shown in our graph, both task-task edges and task-file edges are labeled with stamps (e.g., timestamp, hashsum) that determine if the task output respectively file content is up-to-date.

The incremental build algorithm of Pluto takes the dependency graph of the previous run and selectively reruns tasks to ensure consistency of the build. The dependency graph of a build is consistent if for each invoked task (i) all read and written files are up-to-date and (ii) the outputs of all called tasks are up-to-date. An incremental

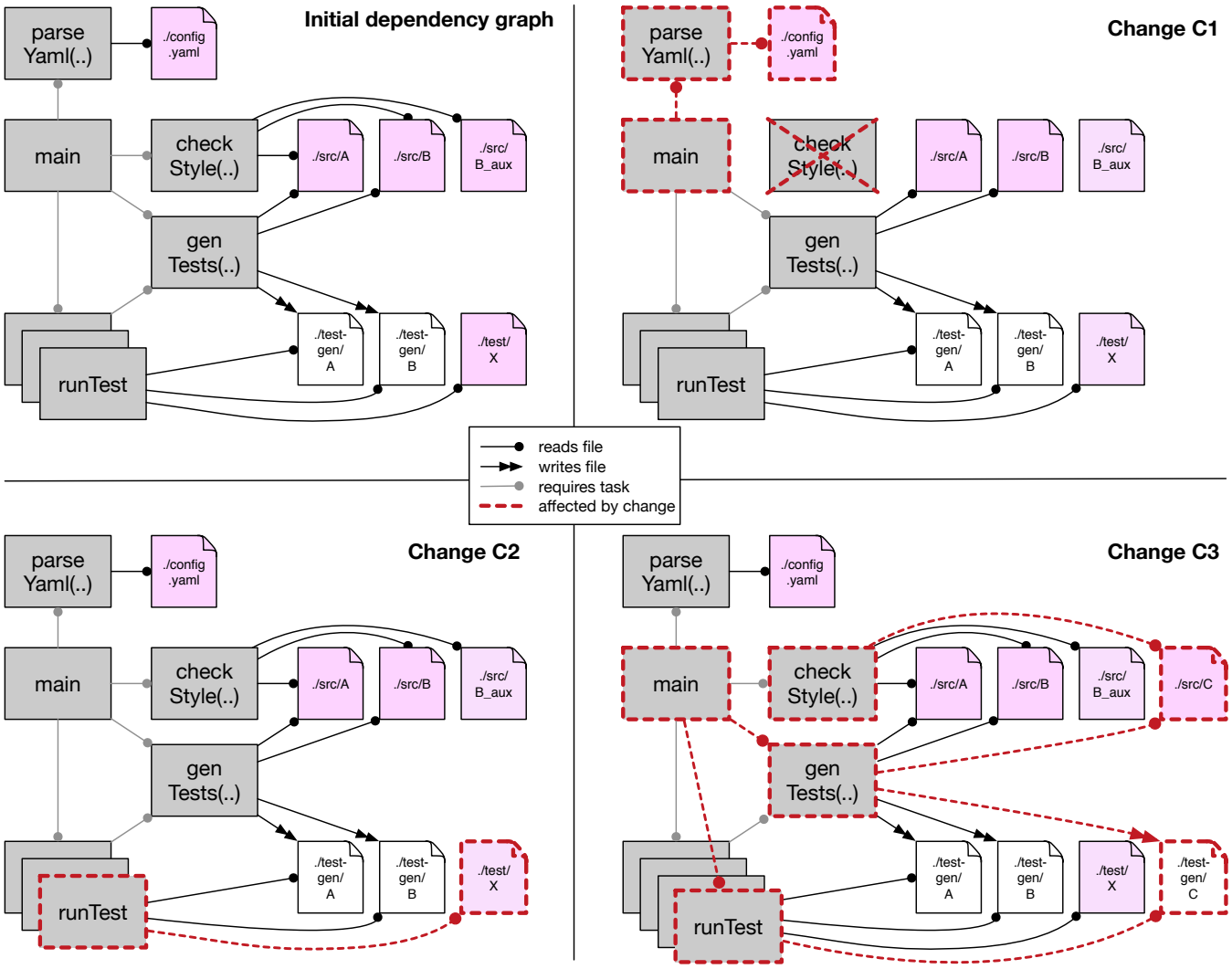

Figure 2: The dependency graph of a build captures task and file dependencies and is the basis for incremental building.

build algorithm is correct if it always restores consistency [3]. Or intuitively: a correct incremental build algorithm yields the same result as a clean build. The challenge is to restore consistency with as little computational effort as possible.

For example, let us assume the initial dependency graph (top-left in Figure 2) is consistent to begin with. We discuss three different changes **C1**–**C3**:

- **C1**: If we change file `config.yaml` to turn off style checking, task `main` becomes inconsistent since the stamp of its file dependency changes (e.g., newer timestamp, changed hashsum). We can restore consistency by rerunning tasks `parseYaml` and `main` only; all other tasks remain consistent since they neither invoke `main` nor read files written by `main`. The incremental build yields a new dependency graph (top-right in Figure 2), where the new invocation of `main` does not depend on `checkStyle` anymore.
- **C2**: If instead, we change the content of user test `./test/X`, task `runTest(./test/X)` becomes inconsistent and needs rerunning. Since task `main` calls `runTest(./test/X)`, it needs rerunning if the value returned by `runTest` changes such that we obtain a different number of `failed` tests. Either way, the dependency graph remains unaffected (bottom-left in Figure 2).
- **C3**: Finally, if we add a source file `./src/C`, tasks `checkStyle` and `genTests` are affected because they depend on the directory `./src`. If the new file has a style error, this affects `main` and yields a new dependency graph where no testing occurs (not shown). Otherwise, let us assume `genTests` produces a new test `./test-gen/C` for the added file, which affects `main`'s scan of directory `test-gen`. During the subsequent rerun of `main`, we discover a new task invocation `runTest(./test-gen/C)` (bottom-right in Figure 2).

The incremental build algorithm of Pluto can handle these and any other changes correctly. Moreover, the algorithm is optimally incremental in the sense that it only executes a task if absolutely necessary. The basic idea of the algorithm is to start at the root node(s) of the dependency graph, to traverse it depth-first, and to interleave consistency checking and rerunning. In particular, while rerunning a task that invokes another task, if the invoked task exists in the dependency graph, continue with consistency checking of the invoked task and only rerun it if necessary. This interleaving of consistency checking and rerunning is what enables support for conditional and iterative task invocations.

Problem Statement. The incremental build algorithm of Pluto has an important limitation. Irrespective of the changed files, it has to traverse the entire dependency graph to check consistency and to discover tasks that need rerunning. While that may be fine for larger changes that affect large parts of the graph like **C3**, the overhead for small-impact changes like **C2** is significant. Especially in interactive settings, where developers routinely trigger sequences of small-impact changes, this overhead can quickly render the system unresponsive. The problem is that the algorithm does not *scale down* to small changes when *scaling up* to large dependency graphs.

Our goal is to design, realize, and evaluate a new incremental build algorithm for programmatic build scripts that scales in the size of a change. That is, rebuild times should be proportional to the impact a change has on the overall build. In particular, rebuild times should be independent of the size of the dependency graph. These requirements preclude a full traversal of the dependency graph to discover affected tasks as done by Pluto. Instead, our new algorithm takes the set of changed files as input and only ever visits affected tasks.

3 KEY IDEA AND CHALLENGES

The key idea to increasing the scalability of the Pluto algorithm is to execute tasks *bottom-up*. In this section, we motivate this approach, and we discuss the corner cases that require adjustments to a pure bottom-up algorithm.

3.1 Bottom-Up Traversal

The key problem of the Pluto build algorithm is that it visits and checks tasks that are ultimately unaffected. For example, in change **C2** in Section 2, only a single task (`runTest`) is affected by the change to file `./test/X`. However, Pluto will visit and check *all* reachable tasks in a top-down depth-first traversal, including the tasks that are not affected by the change (`parseYaml`, `checkStyle`, and `genTests`). Establishing that these tasks *are unaffected* is expensive, as our benchmarks demonstrate (Section 6).

To make the algorithm scale, it should only visit the nodes of the dependency graph that are actually affected by a change. The changes that trigger a re-build are to *files*, which are at the *leaves* of the dependency graph. Tasks that need to be recomputed depend directly or indirectly on such file changes. Instead of looking for tasks that may indirectly depend on a change and gradually getting closer to the actual change, as Pluto does, why not start with those changes and the tasks that depend on them?

The key idea of our algorithm is to *traverse the dependency graph bottom-up*, driven by file changes, only visiting and checking affected tasks. The algorithm first executes the tasks that are *directly* affected by changed files. For example, in change **C2**, file `./test/X` changes, which directly affects task `runTest(./test/X)`, which must therefore be re-executed. Tasks can also be *indirectly* affected by a file change, namely when it reads a file produced by an affected task or when it reads the output value of an affected task. For example, in change **C3**, file `./src/C` is added, which triggers re-execution of `genTests`, which yields a new output value to main, which thus is indirectly affected, re-executes, and creates a new `runTest` task. A subsequent edit of file `./src/C` triggers `genTests` again, which produces the same value as before but updates the generated file `./test-gen/C`, which affects the corresponding `runTest` task (the `main` task is not affected this time).

Thus, a bottom-up traversal executes tasks that are affected by changed files or by other affected tasks, following a path from the changed leaves of the dependency graph to the root(s). However, a pure bottom-up traversal is not adequate to support programmatic build scripts with dynamic dependencies. We discuss the adjustments that are necessary to realize an adequate algorithm.

3.2 Top-Down Initialization

In order to perform a bottom-up traversal over the dependency graph, we need a dependency graph to start with. Therefore, we start with Pluto's top-down algorithm to obtain the initial dependency graph. This is efficient, since every task is affected in the initial build.

3.3 Early Cut-Off

By default, a bottom-up traversal takes the transitive closure of dependencies, re-executing all tasks on the path from a changed file to the root(s) of the dependency graph. However, re-execution of a task does not always lead to a new result. If the result was the same as before, the path to the root can be cut off early. For example, in **C2** `main` depends on `runTest(./test/X)`, which depends on the changed `./test/X` file. So, do we need to re-execute `main`? That depends on the output of task `runTest(./test/X)`. If the result is a different (integer) value than before, the number of tests that fail changes, and `main` should be re-executed Otherwise, `main` is not affected and we can cut off the build early[1], as shown in the bottom-left part of Figure 2.

3.4 Order of Recomputation

Another potential problem of naive bottom-up evaluation is that tasks may be executed multiple times. For example, in change **C3**, `main` depends on two existing affected tasks: `checkStyle` and `genTests`. A possible execution trace when `checkStyle` *does affect* `main` (not shown in the figure), is to execute `checkStyle`, then `main` which is affected by `checkStyle`, then execute `genTests`, and then execute `main` again because it is affected by `genTests`. Executing a task multiple times is not only inefficient, but also causes glitches: inconsistent results that are exposed to users.

[1] In a real-world build script, `runTest` would output a report of which tests fail and why, and `main` would be re-executed whenever this changes. We support this, but chose to keep the example from Section 2 simple for demonstration purposes.

To avoid such re-executions, we should ensure that all affected dependencies of a task are executed before the task itself. Instead of eagerly executing tasks when encountered during a bottom-up traversal, we *schedule* tasks in a priority queue, which is topologically sorted according to the dependency graph. Until the queue is empty, scheduled tasks are removed from the front of the queue and executed. The topological ordering of the priority queue ensures that task dependencies are executed before the task itself.

3.5 Dynamic Dependencies

The final challenge is to support dynamic dependencies during a bottom-up traversal. Consider change **C3** again, where a new task `runTests(./test-gen/C)` is discovered by `main`. A bottom-up traversal can never detect such a dynamic dependency, since it only has access to the dependency graph of the previous run. To remedy this, we temporarily switch to top-down depth-first building when executing a task, so that the task can discover dependencies to new tasks, discover dependencies to existing tasks, or remove existing dependencies.

When discovering a dependency to a new task, top-down depth-first building continues recursively by (eagerly) executing the new task. For example, in change **C3**, task `main` is (indirectly) affected and thus is built in a top-down manner (after its dependencies `checkStyle` and `genTests` have been built), which recursively calls task `runTests(./test-gen/C)` and registers a dependency to it. Furthermore, when a dependency is discovered to a task t that exists in the old dependency graph, it might be affected already. That is, t and dependencies of t may have been scheduled in the queue. We cannot execute t before executing its dependencies. Therefore, we temporarily switch back to bottom-up building, executing dependencies of t that are scheduled in the queue, until t itself is executed or found unaffected. Then we switch back to top-down building and continue executing the caller. Finally, when a dependency is removed (i.e., dependency to a task that was made in the previous run, but not in this run), the dependency graph is updated, but no further action is taken.

3.6 Dependency Graph Validation

As in the Pluto algorithm, we also need to enforce validity of the dependency graph by detecting overlapping generated files, hidden task dependencies, and cyclic tasks. An overlapping generated file occurs when more than one task generates (creates or writes to) the same file. This makes it unclear in which order those tasks must be executed to bring the file into a consistent state, and is therefore disallowed. Furthermore, a hidden dependency occurs when a task requires (reads) a file that was generated by another task, without the requiring task depending on the generator task. Such a dependency must be made explicit, so that the generated file is updated by the generator task before being read by the requiring task. Finally, a task is cyclic when it (indirectly) calls itself. We disallow cyclic tasks to ensure termination of the build algorithm. We check these invariants on-the-fly while constructing the new dependency graph for subsequent incremental builds.

```
1:  var T_q; var T_e; var O_c; var DG_new
2:  function buildNewTask(t, DG_old)
3:      T_e := ∅; O_c := ∅; DG_new := DG_old
4:      exec(t)
5:  function buildWithChangedFiles(F, DG_old)
6:      T_e := ∅; O_c := ∅; DG_new := DG_old
7:      T_q := new PriorityQueue(DG_old.depOrder())
8:      schedAffByFiles(F, DG_old)
9:      while T_q ≠ ∅ do
10:         execAndSchedule(T_q.poll(), DG_old)
```

Figure 3: Algorithm: Main build functions.

```
1:  function execAndSchedule(t, DG_old)
2:      val r := exec(t)
3:      schedAffByFiles(r.genFiles, DG_old)
4:      schedAffCallersOf(t, r.output, DG_old)
5:      return r.output
6:  function schedAffByFiles(F, DG_old)
7:      for f ← F do
8:          for (stamp, t) ← DG_old.requireesOf(f) do
9:              if ¬stamp.isConsistent(f) then
10:                 T_q := T_q ∪ t
11:         if (stamp, t) ← DG_old.generatorOf(f) then
12:             if ¬stamp.isConsistent(f) then
13:                 T_q := T_q ∪ t
14: function schedAffCallersOf(t, o, DG_old)
15:     for (stamp, t_call) ← DG_old.callersOf(t) do
16:         if ¬stamp.isConsistent(o) then
17:             T_q := T_q ∪ t_call
```

Figure 4: Algorithm: Bottom-up Building.

```
1:  function exec(t)
2:      if t ∈ T_e then abort
3:      T_e := T_e ∪ t; val r := t.run(); T_e := T_e \ t
4:      DG_new := DG_new ∪ r; validate(t, r); observe(t, r.output)
5:      O_c[t] := r.output; return r.output
6:  function require(t, DG_old)
7:      if o ← O_c[t] then return o
8:      else if t ∈ DG_old then return requireNow(t, DG_old)
9:      else return exec(t)
10: function requireNow(t, DG_old)
11:     while val t_min := T_q.leastDepFromOrEq(t) do
12:         T_q := T_q \ t_min
13:         val o := execAndSchedule(t_min, DG_old)
14:         if t = t_min then return o
15:     val o := DG_old.outputOf(t)
16:     observe(t, o); O_c[t] := o; return o
17: function validate(t, r)
18:     for f ← r.genFiles do
19:         for (_, t_gen) ← DG_new.generatorOf(f) do
20:             if t ≠ t_gen then abort
21:     for f ← r.reqFiles do
22:         for (_, t_gen) ← DG_new.generatorOf(f) do
23:             if ¬DG_new.callsTaskTr(t, t_gen) then abort
```

Figure 5: Algorithm: Execution, Requirement, and Validation.

4 CHANGE-DRIVEN INCREMENTAL BUILDING

In this section, we present our *hybrid algorithm* that mixes bottom-up and top-down incremental building based on the observations and ideas from the previous section. We present the algorithm in three parts: main functions (Figure 3), bottom-up building (Figure 4), and execution (Figure 5). All functions share four global variables defined at the top of Figure 3. Variable T_q is a topologically ordered priority queue of affected tasks that still need to be executed. Variable T_e is a set of currently executing tasks, used to detect cyclic tasks. Variable O_c is a cache of output values for tasks that have already been executed. Finally, variable DG_{new} is the new dependency graph that is constructed from the old dependency graph and dynamic dependencies on-the-fly.

We provide two entry points to incremental building in Figure 3, both of which first clear the set of executing tasks T_e, clear the cache O_c, and copy the old dependency graph DG_{old} to DG_{new}. Function `buildNewTask` is the entry point for an initial build. Function `buildNewTask` then simply invokes function `exec` (Figure 5) to execute the task. We describe `exec` below.

The second entry point `buildWithChangedFiles` is more interesting as it initiates bottom-up building. It takes as input a set of changed file paths F, represented as filesystem path strings such as `./config.yaml`, and the old dependency graph DG_{old}. The basic idea is to schedule and run affected tasks using priority queue T_q until all affected tasks are up-to-date. To this end, we create a new priority queue using the task dependencies in DG_{old} as a topological ordering. We call function `schedAffByFiles` (described below) with the old dependency graph to find all tasks *directly* affected by the changed file paths F, and add those tasks to the queue T_q. The main loop of bottom-up building is the following *while*-loop: As long as there are affected tasks in the queue, poll a scheduled task (retrieve the task at the front and remove it) from the queue, execute it, and add all tasks affected by it to the queue. Since the queue is topologically ordered, dependencies of tasks are executed before the task itself. Unless a task itself does not terminate (for example by recursively calling new tasks ad infinitum), the queue becomes empty at some point since cyclic tasks are disallowed, terminating the algorithm.

4.1 Bottom-Up Building

Whenever a task occurs in the priority queue T_q, it is definitely affected (directly or indirectly) by changed files. Hence, no further consistency check is necessary. Function `execAndSchedule` in Figure 4 accepts an affected task, runs it unconditionally using `exec`, and schedules new tasks based on the generated files and output value of the executed task. If a task does not change or create new generated files, nor produce a new output value, no new tasks will be scheduled and building may be cut off early.

Function `schedAffByFiles` schedules tasks based on changed file paths F. If task t requires a changed file and the stamp *stamp* has changed (is inconsistent) then t is affected by the change to f and is scheduled by adding it to T_q. Analogously, if a task generates a file that has changed, it is affected and thus scheduled. A stamp contains a summary of a file's content, such as the last modification date or a hash, and is used to efficiently check whether a file has changed with `isConsistent`. For example, when using the file's modification date as a stamp, we compare the modification date in the stamp, with the current modification date of the file on the local filesystem, and consider the file changed if the modification date is different. We use the old dependency graph DG_{old} (computed in a previous run of the algorithm) to find tasks that require a file (`requireesOf`), and to find the task that generates a file (`generatorOf`), along with the stamp that was produced at the time the dependency was created.

Likewise, the `schedAffCallersOf` function schedules callers of task t based on changes to its output value o. If t_{call} has a dependency to task t, and that dependency is inconsistent with relation to the new output value o of t, then t_{call} is affected by the new output value o and is scheduled. Similarly, we use a *stamp* of the output value, which could be the full output value, such as an integer representing the number of failing tests, or a summary of the value such as a hash, and compare the stamp with the new output value with `isConsistent`. Finally, the old dependency graph DG_{old} is used to find callers of a task with `callersOf`.

4.2 Execution, Requirement, and Validation

Function `exec` (Figure 5) executes the body of t. During task execution, a task may require (call) other tasks with the `require` function. Therefore, we first need to check if we are already executing task t, and abort when a cycle is detected. Then, we add t to the set of executing tasks T_e, `run` the body of the task, and remove t from T_e. Once execution completes, we update the new dependency graph DG_{new} with the result r of executing t. A result r contains the dynamic dependencies the task made during execution: a set $reqFiles$ of read files, $genFiles$ of created or written to files, and a set $reqTasks$ of other tasks that were called by t; and the output value *output* that the task produced. A dependency graph DG is a set of those results, where each task has a single result. We then `validate` the new dependency graph, call any external observers of the task's output with `observe`, cache the output, and finally return the output.

We use function `exec` to execute tasks both during bottom-up and top-down traversals. While `exec` is agnostic to the traversal order, function `require` must take care to handle tasks required bottom-up and top-down correctly. We distinguish three cases. If t was already executed (visited) this run, we return its cached output value $O_c[t]$. Otherwise, we check if t was in the old dependency graph DG_{old}. If task t is new and does *not* occur in DG_{old}, then we execute it unconditionally. Note that no existing task in DG_{old} can depend or be affected by the new task t.

If task t existed before in DG_{old}, we only execute it if it is actually affected. Since the caller of t awaits the output of t, we use function `requireNow` to force its checking and possible execution *now*. Task t is affected if it occurs in queue T_q or if any of its dependencies occurring in T_q will affect it later. Function `requireNow` repeatedly finds dependency t_{min} of t that is lowest in the dependency graph (closes to the leaves). Since the queue only contains affected tasks, we execute t_{min} and schedule tasks affected by it. We continue until either we have executed the required task t, or until no more dependencies of task t are affected and we can reuse t's output value from the old dependency graph with DG_{old}.`outputOf`(t). Note that this latter case always triggers for tasks scheduled bottom-up

by `buildWithChangedFiles`, because their dependencies cannot occur in T_q anymore.

The `validate` function incrementally validates the correctness of the new dependency graph after executing a task t. For a dependency graph to be correct, it may not have overlapping generated files, nor any hidden dependencies. If another task t_{gen} generates the same file f as t does, there is an overlapping generated file and execution is aborted. Furthermore, if t requires a file f without a (transitive) task dependency on t_{gen} that generates f, there is a hidden dependency and execution is aborted. In both cases, this signals that there is an error in the build script.

4.3 Properties

An incremental build algorithm is correct if it produces the exact same result as a clean build. Therefore, all affected and new tasks must be executed. Our algorithm is correct for tasks in the old dependency graph: if a task is affected, it will be scheduled. A task is affected directly by depending on a changed file, or indirectly (transitively) by depending on a changed file that an affected task generates, or by depending on the changed output of an affected task. All indirectly affected tasks are always found by traversing the dependency graph bottom-up, through polling the queue and scheduling affected tasks. Finally, all scheduled tasks are executed.

Our algorithm is also correct for new tasks that are executed top-down like the Pluto algorithm, which is correct [3]. The only difference is the `reqireNow` function which first executes the dependencies of the task, but does eventually execute the task itself. Therefore, the hybrid algorithm is correct.

For optimality, we only consider and execute affected tasks. For existing tasks, this is true because only affected tasks scheduled. New tasks are affected and always executed. However, we only want to execute *needed* tasks. The hybrid algorithm considers all task in the old dependency graph as needed. This is an overapproximation, because it can happen that an affected task is not needed any more after top-down execution, since a task may remove its dependency to an affected task. Therefore, theoretically, the hybrid algorithm is only partially optimal. However, this is a rare case, as shown in the evaluation in Section 6.

5 IMPLEMENTATION

We have implemented the hybrid algorithm as an alternative execution algorithm for PIE [16], a system for developing interactive software development pipelines, consisting of a DSL and API for implementing interactive pipelines, and a runtime for incrementally executing them. Interactive software development pipelines are similar to incremental build systems: they are used to incrementally build software artifacts, and also require fast feedback for usage in interactive environments with many low-impact changes such as IDEs and code editors. PIE builds forth on Pluto by reusing its model and algorithm, but provides a concise and expressive DSL for developing interactive pipelines and build scripts, minimizing boilerplate in contrast to Pluto's Java API.

Our algorithm is implemented as a separate executor in the PIE runtime, fully conforming to its API. That is, we can run existing PIE build scripts without changes to our algorithm. Furthermore, since PIE implements the Pluto build algorithm, we can compare our algorithm against Pluto's, for the exact same build scripts. PIE, including our hybrid algorithm, is open source software that can be found online[2].

6 EVALUATION

In this section, we evaluate the performance of the hybrid algorithm, compared to Pluto's pure top-down algorithm. We describe our experimental setup, show the results, interpret them, and discuss threats to validity.

6.1 Experimental Setup

We compare the performance of the Pluto incremental build algorithm, as implemented in the PIE runtime, against our hybrid incremental build algorithm, which we have implemented in PIE runtime.

Build Script. As a build script, we reuse the Spoofax-PIE pipeline, a reimplementation of a large part of the Spoofax pipeline, which was used as a case study of PIE [16]. Spoofax [13] is a language workbench [4] (a set of tools for developing languages) in which languages are specified in terms of meta-languages, such as SDF [27] for syntax specification, and NaBL [15, 23, 26] for name and type analysis. The Spoofax pipeline derives artifacts from a language specification, such as a parse table for parsing, and a constraint generator and solver for solving name and type analysis. Furthermore, Spoofax supports interactive language development in an IDE setting, enabling a language developer to modify a language specification, resulting in immediate feedback in example programs of that language, and also supports developing multiple languages side-by-side. The Spoofax-PIE reimplementation supports these features. The build script is open-source and can be found online [3].

As input, the Spoofax build script takes a workspace directory consisting of language specifications, where each language specification has a configuration file describing how to build the language specification, a specification of the syntax, styling, and name and type analysis in meta-languages, and example programs. A configuration file at the root of the workspace lists the locations of all language specifications, and locations of the Spoofax meta-languages. As a concrete workspace, we use a directory with three Spoofax language specifications for the Tiger, Calc, and MiniJava languages.

Describing the Spoofax build script is outside of the scope of this paper. However, we do argue why Spoofax requires a programmatic build script with dynamic dependencies. The Spoofax build script frequently makes use of conditional building, where the result of executing a task influences a condition for another task. For example, when a program fails to parse, the program cannot be analyzed, since analysis requires an AST. Therefore, a condition that checks whether the parsing task succeeds guards the analysis task. Furthermore, Spoofax also makes frequent use of iterative building, where tasks are invoked multiples on different inputs which are outputs of previous tasks. For example, there is a single task description for parsing a file, which is dispatched based on the result of parsing the workspace configuration file, parsing the language specification configuration files, the concrete files that are in the workspace, and the extension of each file. Without a

[2] https://github.com/metaborg/pie
[3] https://github.com/metaborg/spoofax-pie

programmatic build script, all these forms of variability would have to be encoded in the configuration step of a declarative build script, which is not possible because many values only become evident during build script execution.

Changes. To measure incremental performance, we have synthesized a chain of 60 realistic changes with varying impacts. First, a from-scratch build is performed that builds all language specifications. Then, we make changes in the form of opening or changing a text editor, requiring execution of a task that provides feedback for that editor, or of modifying and saving a file, which requires execution of tasks that keep the workspace up-to-date.

Changes include: editing and saving example programs, styling specifications, syntax specifications, and name and type analysis specifications; adding a new language specification; undoing changes; and two extreme cases where we run the build with no or all files changed. These changes have varying impacts, where the impact is determined by how many tasks are affected by a change, and the run time of those tasks. For example, changing a syntax definition file requires recompilation of the parse table and reparsing of all example programs. Changing the name and types specification has a larger impact because it requires regeneration of a constraint generator, compilation of the constraint generator, application of the generator against all example programs, and finally application of the constraint solver to solve all generated constraints. A small impact change is editing an example program in an editor, which just requires parsing, styling, and analysis for that program.

We run the exact same changes against the Pluto and our hybrid algorithm, with the only difference that we pass the changed files to our hybrid algorithm, whereas Pluto does not require this. We run the chain of changes against one algorithm in one go, to simulate a full editing session.

Technicalities. We run the benchmark using the JMH [1] benchmarking framework, which runs the benchmark for an algorithm in a separate forked JVM, letting the JVM JIT fully specialize to that algorithm. Furthermore, it runs the benchmark multiple times before starting measurements, to ensure that the JVM is warmed up. Finally, it ensures that the garbage collector is executed before running a benchmark, so that garbage produced in a previous run does not influence the new one.

We have executed the benchmark on a MacBook Pro with a 2.7 GHz Intel Core i7 processor, 16 GB of 1600 MHz DDR3 memory, and a SSD, running macOS 10.12.6. The benchmark was executed with a 64-bit JRE of version 1.8.0b144, with 16 MB of stack memory, and 4 GB of heap memory.

6.2 Results and Interpretation

We now show the benchmarking results and interpret them. It is not possible to discuss time measurements for all 60 changes. Therefore we aggregate the time taken for different kinds of changes and show those instead. Figure 6 shows the time measurements for each aggregated change, for both the Pluto and hybrid algorithm, in a column chart with logarithmic scale. We now go over the results for each kind of change.

A) Initial build. First, we perform an initial build, building all language specifications. To obtain the initial dependency graph, we use top-down building. Therefore, both the Pluto and hybrid algorithm perform identically.

B) Editor changes. We aggregate the running time for all editor changes: both opening new editors and editor text changes, for example programs and language specifications. For all editor changes combined, the hybrid algorithm is 11 seconds faster, providing a speedup of 1016%. The speedup is high because these changes have a small impact, and therefore are efficiently handled by the hybrid algorithm. It is important to quickly process editor changes in IDEs, as programmers make many changes to editors and require fast feedback cycles.

C) Example program file changes. We modify the files of several example program, and add a new example program file. For these changes combined, the hybrid algorithm takes 0.005 seconds, providing a 9 second (182133%) speedup. Again, these changes have a small impact, and are therefore efficiently handled by the hybrid algorithm, whereas the Pluto algorithm still requires checking of the entire dependency graph.

D) Styling specification change. We modify the styling specification of the Calc language, and add a styling specification to the Tiger language. For these changes, the hybrid algorithm is 8 seconds faster, providing a 1245% speedup. The impact of these changes are slightly larger: changes to the styling specification require restyling of open editors, but are still relatively smaller in impact and thus efficiently handled.

E) Adding language specification. We add the MiniJava language to the workspace, requiring it to be built, and its example programs to be processed. Since this change causes many new tasks to be executed, its impact is large. The hybrid algorithm performs roughly the same as the Pluto algorithm, providing only a 2.9 second (11%) speedup because of reduced dependency graph checking.

F) Syntax specification small change. We modify lexical syntax definition of the Calc language to parse numbers incorrectly, and undo the change afterwards. This requires the parse table to be rebuilt, and requires processing of Calc's example programs. The hybrid algorithm provides a 4.5 second (118%) speedup, because a smaller part of the dependency graph is checked.

G) Syntax specification cascading change. We modify the Calc syntax definition in such a way that the resulting parser will fail to parse all example programs, and also in such a way that new AST signatures need to be generated. From the syntax specification, Spoofax generates AST signature files that the name and type analysis uses. These signature files have changed, therefore requiring the name and type analysis specification to be recompiled. Finally, all example programs must be reparsed and reanalyzed.

However, because example programs cannot be parsed any more, they also cannot be analyzed any more, since name and type analysis requires an AST. Therefore, the dependency from the process example file task, to the task that analyzes the AST of an example program, disappears. The Pluto algorithm first visits the process example file task, which removes its dependency to the analysis task, and therefore never recompiles the name and type analysis specification. However, the hybrid algorithm goes bottom-up to first recompile the name and type analysis specification, and only then executes the process example file task, therefore executing a

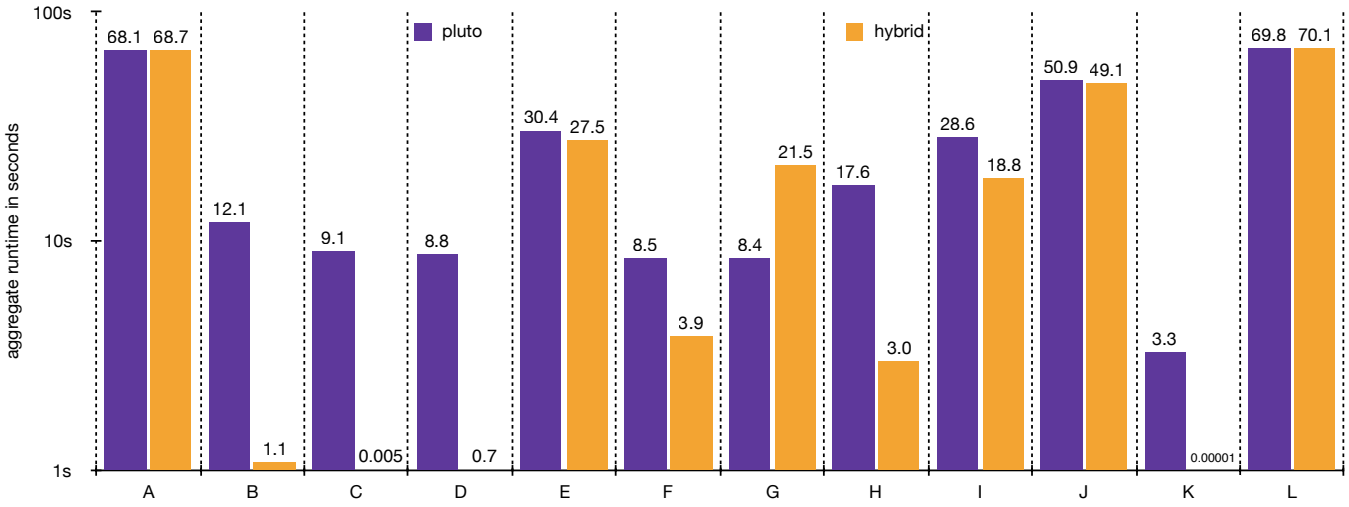

Figure 6: Column chart with aggregate benchmark time measurements. The x-axis represent the different changes (described below), the y-axis represents the time taken in seconds in logarithmic scale. For each change, we show the time taken for the Pluto algorithm and our hybrid algorithm. A = initial build, B = all editor changes, C = example program changes, D = styling specification changes, E = adding language specification, F = syntax specification small change, G = syntax specification cascading change, H = syntax specification refactor, I = analysis specification changes, J = analysis specification refactor, K = no changes, L = all files changed.

task that was not required to be executed. In this case, the hybrid algorithm was 13 seconds slower, causing a 61% slowdown.

This is a tradeoff of the hybrid algorithm: if a dependency to a task disappears, the hybrid algorithm will still visit it. However, these cases are very rare, only a single change triggers this kind of behavior. For example, if at least one example program could be parsed into an AST (possibly through error recovery), the analysis specification has to be recompiled. We undo the change afterwards to make example programs parse again.

H) Syntax specification refactor. We refactor a part of the MiniJava syntax definition into another file, which results in a semantically equivalent parser. The hybrid algorithm provides a 14.5 second (483%) speedup, because it first rebuilds the parse table, detects that it did not change, and then cuts off the build early. Contrary to the previous change, a bottom-up traversal here helps in cutting down the incremental build time, by not even traversing the unaffected part of the dependency graph.

I) Analysis specification change. We modify the name and type analysis specification of the Calc language, such that it scopes bindings differently, and undo the change afterwards. Because changing these specifications has a moderate impact, the hybrid algorithm provides a moderate 9.7 second speedup (52%).

J) Analysis specification refactor. We refactor a part of the Tiger name and type analysis specification into another file. Even though this results in a semantically equivalent analyzer, the change detection of the Spoofax-PIE build script is not smart enough to detect this. Because compiling the name and type analysis specification, and then performing constraint solving for all Tiger example programs, is expensive, this change has a large impact. Therefore, the Pluto and hybrid algorithm perform nearly identically.

K) No changes. When there are no changes, the hybrid algorithm essentially performs no work, completing in sub-millisecond time, while the Pluto algorithm still needs to check the entire dependency graph, costing 3.3 seconds of time. This is the constant overhead that even small-impact changes suffer from with the Pluto algorithm, which the hybrid algorithm saves.

L) All files changed. Finally, we change all source files by appending a space to the end of each file. Realistically, this kind of change can happen when checking out a different branch in a source control management system such as Git. When all source files change, using a bottom-up approach makes no sense, since (almost all) tasks will be affected, while incurring overhead because of scheduling. Therefore, we detect when more than 50% of source files (all required files, for which there is no generator task) change, and run a top-down build with the Pluto algorithm instead, therefore running as fast as the Pluto algorithm does. This heuristic seems to work well, but may require further tweaking.

Conclusion. We can conclude that, for this build script and workspace directory, our algorithm scales better with the impact of a change than the Pluto algorithm, for many kinds of changes. The only exceptions being when all files are changed, for which a full rebuild could be triggered, or when a dependency to an expensive task is removed, which rarely happens.

6.3 Threats to Validity

A possible threat to validity is that we have benchmarked the algorithms against a single build script. However, it is a complex build script that represents the realistic scenario of interactive language development in a language workbench. For example, the build script

requires dynamic file dependencies in order to track precise dependencies which only become evident during a build. Furthermore, it also requires dynamic task dependencies, in order to dispatch the correct tasks based on the configuration of the workspace and each language specification.

Another possible threat is that we have synthesized a chain of changes, instead of using existing change scenarios. However, we have constructed 60 changes to different kinds of aspects; such as changing an example program, and changing a file of the syntax specification; and with varying levels of impact, ranging from changing the text in a single editor, to changing a file of the name and type specification, which transitively affects many other tasks.

7 RELATED WORK

There is a large body of work on incremental build systems. Make [25] is an incremental build system with declarative build rules based on files. It has limited support for dynamic file dependencies, and no proper support for dynamic task dependencies. Because of these limitations, Make scales well for simple build scripts, since it can first topologically sort dependencies, iterate over the dependencies, and incrementally execute affected tasks. While it is possible to emulate dynamic task dependencies, this requires tedious Makefile generation, encoding of all dependencies as files, and recursive Make execution. Therefore, Make is not sufficient for more complicated build scripts.

Many build systems follow a similar approach to Make, first building a task DAG, and then executing it. For example, Gradle [12], Bazel [6], Buck [5], PROM [14], Fabricate [11], Tup [24], and Ninja [18] follow this approach, with slight variations. Gradle is a build automation tool, programmable in the Groovy language, that, like our hybrid algorithm, also supports values as inputs and outputs of tasks. PROM replaces declarative make rules with logical programming, while keeping the same incremental build algorithm. Fabricate uses system tracing to automatically infer file dependencies, but is only supported on Linux. Tup, like our hybrid build algorithm, requires a list of changed files as input, instead of scanning all files, to more efficiently build the task DAG. Ninja, unlike Make, detects changes to the commands of a rule, resulting in a rebuild if the rule is changed. None of the above systems supports dynamic file or task dependencies.

Some build systems intertwine incremental execution with the discovery of file and task dependencies. Pluto [3] is a Java library for developing incremental build scripts with dynamic dependencies. As discussed throughout this paper, Pluto uses a top-down algorithm that does not scale to small changes over large dependency graphs. OMake [10] is a build system with Make-like syntax, but with a richer dependency tracking mechanism and a more complicated algorithm. It has limited support for dynamic file dependencies through scanner rules that scan depfiles and register their dependencies during execution. However, it does not support dynamic task dependencies; all tasks dependencies are specified statically in the build rules. Shake [21, 22] is a Haskell library for implementing build scripts with incremental execution. It has limited support for dynamic file dependencies, allowing needed files to be discovered dynamically, but generated file dependencies must be specified statically as the build target. It also has limited support for dynamic task dependencies: Tasks are named by keys, and those tasks can be required like files through their keys. However, these tasks are not parameterized, nor can they return values, making their use as dynamic task dependencies tedious.

Our work is also related to approaches on incremental computing. Datalog [2] is a logic programming language with incremental solvers [7]. The are several differences between our hybrid algorithm and incremental Datalog solvers. Datalog solvers can deal with cycles, eagerly compute all facts, and use static dependencies from the Datalog program, whereas the hybrid algorithm (and build systems in general) disallow cycles, only compute demanded facts, and use dynamic dependencies.

Adapton [8, 9] is a library for on-demand (lazy) incremental computation. Like the Pluto and our hybrid algorithm, Adapton supports a form of dynamic task dependencies: dynamic computation dependencies which form a computation graph. Initially, Adapton builds a full computation graph. Then, to achieve incrementality, when a node in the computation graph is affected by a change, it transitively marks all dependent nodes by setting a *dirty flag*. Then, when a computation result is demanded, it transitively reruns all dirty nodes that are required by the computation. The downside of dirty flagging is that it is an over-approximation of what actually needs to be executed, ignoring cases where the output of a computation does not change. In build systems, where many tasks may depend on a single compiler task, and where computations include calling compilers that can run for seconds to even minutes, avoiding recomputation when dependencies do not change, is crucial. For example, when we change a syntax specification, but the resulting parse table does not change, dirty flagging has already marked parsing all example files as dirty. Therefore, our hybrid algorithm checks outputs of task to cut off the build early, instead of performing dirty flagging and propagation.

8 CONCLUSION

We have shown the need for an efficient, precise, and expressive build system. Many build systems are efficient and precise, but not expressive, making complex build script development tedious. Pluto, a recent incremental build system that supports programmable build scripts with dynamic dependencies, is expressive, but does not scale with the impact of a change, because it requires a top-down traversal over the entire dependency graph for each change. To overcome this scalability problem, we have realized a hybrid algorithm that mixes bottom-up building for scalability, and top-down building for expressiveness through dynamic dependencies. We have evaluated the performance of our hybrid algorithm against Pluto's algorithm, with a case study on the Spoofax language workbench. The evaluation demonstrates that the hybrid algorithm, with the exception of one kind of change, indeed scales better with the impact of a change, and is therefore faster than the Pluto algorithm, in particular for low-impact changes.

ACKNOWLEDGEMENTS

This research was supported by NWO/EW Free Competition Project 612.001.114 (Deep Integration of Domain-Specific Languages) and NWO VICI Project (639.023.206) (Language Designer's Workbench).

REFERENCES

[1] [n. d.]. Java Microbenchmarking Harness (JMH). http://openjdk.java.net/projects/code-tools/jmh/. [Online; accessed 27-April-2018].

[2] S. Ceri, G. Gottlob, and L. Tanca. 1989. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Trans. on Knowl. and Data Eng.* 1, 1 (March 1989), 146–166. https://doi.org/10.1109/69.43410

[3] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. 2015. A sound and optimal incremental build system with dynamic dependencies. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 89–106. https://doi.org/10.1145/2814270.2814316

[4] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches - Conclusions from the Language Workbench Challenge. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26-28, 2013. Proceedings (Lecture Notes in Computer Science)*, Martin Erwig, Richard F. Paige, and Eric Van Wyk (Eds.), Vol. 8225. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[5] Facebook. [n. d.]. Buck: a fast build tool. https://buckbuild.com/. [Online; accessed 27-April-2018].

[6] Google. [n. d.]. Bazel - a fast, scalable, multi-language and extensible build system. https://bazel.build/. [Online; accessed 27-April-2018].

[7] Ashish Gupta and Inderpal Singh Mumick. 1995. Maintenance of Materialized Views: Problems, Techniques, and Applications. *IEEE Data Eng. Bull.* 18, 2 (1995), 3–18. http://sites.computer.org/debull/95JUN-CD.pdf

[8] Matthew A. Hammer, Joshua Dunfield, Kyle Headley, Nicholas Labich, Jeffrey S. Foster, Michael W. Hicks, and David Van Horn. 2015. Incremental computation with names. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SPLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, Jonathan Aldrich and Patrick Eugster (Eds.). ACM, 748–766. https://doi.org/10.1145/2814270.2814305

[9] Matthew A. Hammer, Yit Phang Khoo, Michael Hicks, and Jeffrey S. Foster. 2014. Adapton: composable, demand-driven incremental computation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, Michael F. P. O'Boyle and Keshav Pingali (Eds.). ACM, 18. https://doi.org/10.1145/2594291.2594324

[10] Jason Hickey and Aleksey Nogin. 2006. OMake: Designing a Scalable Build Process. In *Fundamental Approaches to Software Engineering*, Luciano Baresi and Reiko Heckel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 63–78. https://doi.org/10.1007/11693017_7

[11] B. Hoyts and Simon Alford. 2009. fabricate. https://github.com/SimonAlfie/fabricate. [Online; accessed 27-April-2018].

[12] Gradle Inc. [n. d.]. Gradle Build Tool. https://gradle.org/. [Online; accessed 27-April-2018].

[13] Lennart C. L. Kats and Eelco Visser. 2010. The Spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010*, William R. Cook, Siobhán Clarke, and Martin C. Rinard (Eds.). ACM, Reno/Tahoe, Nevada, 444–463. https://doi.org/10.1145/1869459.1869497

[14] Thilo Kielmann. 1991. *PROM: A flexible, PROLOG-based make tool.* Technical Report Report TI-4/91. Institute of Theoretical Computer Science, Darmstadt University of Technology, Darmstadt, Germany. http://www.few.vu.nl/~kielmann/papers/THD-SP-1991-04.pdf

[15] Gabriël Konat, Lennart C. L. Kats, Guido Wachsmuth, and Eelco Visser. 2012. Declarative Name Binding and Scope Rules. In *Software Language Engineering, 5th International Conference, SLE 2012, Dresden, Germany, September 26-28, 2012, Revised Selected Papers (Lecture Notes in Computer Science)*, Krzysztof Czarnecki and Görel Hedin (Eds.), Vol. 7745. Springer, 311–331. https://doi.org/10.1007/978-3-642-36089-3_18

[16] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. 2018. PIE: A Domain-Specific Language for Interactive Software Development Pipelines. *The Art, Science, and Engineering of Programming* 2, 3 (2018). https://doi.org/10.22152/programming-journal.org/2018/2/9

[17] Epperly T Kumfert G. 2002. *Software in the DOE: The Hidden Overhead of "The Build".* Technical Report. Lawrence Livermore National Laboratory.

[18] Evan Martin. [n. d.]. The Ninja build system. https://ninja-build.org/manual.html. [Online; accessed 27-April-2018].

[19] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. 2010. The evolution of ANT build systems. In *Proceedings of the 7th International Working Conference on Mining Software Repositories, MSR 2010 (Co-located with ICSE), Cape Town, South Africa, May 2-3, 2010, Proceedings*, Jim Whitehead and Thomas Zimmermann (Eds.). IEEE, 42–51. https://doi.org/10.1109/MSR.2010.5463341

[20] Peter Miller. [n. d.]. Recursive make considered harmful. ([n. d.]).

[21] Neil Mitchell. 2012. Shake before building: replacing make with haskell. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, Peter Thiemann and Robby Bruce Findler (Eds.). ACM, 55–66. https://doi.org/10.1145/2364527.2364538

[22] Andrey Mokhov, Neil Mitchell, Simon Peyton Jones, and Simon Marlow. 2016. Non-recursive make considered harmful: build systems at scale. In *Proceedings of the 9th International Symposium on Haskell, Haskell 2016, Nara, Japan, September 22-23, 2016*, Geoffrey Mainland (Ed.). ACM, 170–181. https://doi.org/10.1145/2976002.2976011

[23] Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2015. A Theory of Name Resolution. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings (Lecture Notes in Computer Science)*, Jan Vitek (Ed.), Vol. 9032. Springer, 205–231. https://doi.org/10.1007/978-3-662-46669-8_9

[24] Mike Shal. 2009. Build System Rules and Algorithms. http://gittup.org/tup/build_system_rules_and_algorithms.pdf. [Online; accessed 27-April-2018].

[25] Richard M. Stallman, Roland McGrath, and Paul D. Smith. 2016. GNU Make manual. Free Software Foundation.

[26] Hendrik van Antwerpen, Pierre Néron, Andrew P. Tolmach, Eelco Visser, and Guido Wachsmuth. 2016. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, Martin Erwig and Tiark Rompf (Eds.). ACM, 49–60. https://doi.org/10.1145/2847538.2847543

[27] Eelco Visser. 1997. *Syntax Definition for Language Prototyping.* Ph.D. Dissertation. University of Amsterdam. Advisor(s) Paul Klint.