Precise, Efficient, and Expressive Incremental Build Scripts with PIE

Gabriël Konat[1] g.d.p.konat@tudelft.nl

Roelof Sol[1] roelofsol@gmail.com

Sebastian Erdweg[2] erdweg@uni-mainz.de

Eelco Visser[1] e.visser@tudelft.nl

[1]Delft University of Technology, [2]Johannes Gutenberg University Mainz

Introduction Build systems check, compile, test, and deploy virtually every software project. Besides increasing software reliability, *incremental* build systems also speed up software development by enabling short feedback cycles. After changing a source file or configuration flag, only a *subset* of the *build tasks* needs to be re-executed to bring the project into a consistent state again. For build tasks that were not affected by a change, the previous result is reused. However, the reliable and long-term maintainable usage of incremental build systems require the following three properties:

Precision Incremental builds must yield the same result as a clean build (correctness), while re-executing the least number of tasks possible (minimality). Therefore, build systems must track *precise dependency information* for each *build task*: which files did it read from or write to, and which other tasks did it require?

Efficiency Rebuilds must be *efficient*. That is, rebuild times must be proportional to the *impact of the change*. Even for large software projects which require a lot of dependency tracking, a small change that only affects a few build tasks should incur a short rebuild time.

Expressiveness Like all software artifacts, build scripts grow during a project's lifetime [7], and require increasing maintenance [6]. Therefore, build scripts must be written in expressive languages that minimise accidental complexity. Build script languages should not require complicated design pattern for expressing common scenarios.

Current incremental build systems focus on efficiency and precision, but lack expressiveness. To support efficient incremental rebuilds, current build systems impose a strict separation of the *configuration* and *build* stages. That is, all build tasks and dependencies (i.e., the variability of the build) are declared in the configuration stage to create a build plan, which is then executed in the build stage. This model contradicts reality, where *how to build an artifact* depends on the execution of other build tasks.

Common workarounds required in this model are over-approximation (`*.h` in Make to depend on all C header files), under-approximation (`own.h` to depend only on your own header file), or additional staging (e.g., Makefile generation and recursive Make [8]). However, these workarounds are not precise: over-approximation is not minimal and under-approximation is unsound. Furthermore, additional staging is not expressive nor maintainable, as it introduces accidental complexity into build scripts. To solve this problem, we need to eliminate staging and instead provide build engineers with an *expressive programming language* for writing build scripts, and support precise dependency tracking through *dynamic dependencies*.

```
taskdef main() -> string {
  val config = parseYaml(./config.yaml);
  if(config.checkStyle && !checkStyle(config.srcDir))
    return "style error";
  val genTests = genTests(config.srcDir, ./test-gen);
  var failed = 0;
  for(test <- ./test/** ++ genTests)
    if(!runTest(test)) failed += 1;
  return "Failed tests: " + failed;
}
taskdef parseYaml(file: path) -> Config { ...
  requires file; ... }
taskdef checkStyle(src: path) -> bool { ... }
taskdef genTests(src: path, dst: path) -> path* { ...
  provides $dst/$src; ... }
taskdef runTest(test: path) -> bool { ... }
```

Listing 1: PIE build script that optionally checks code style, generates tests, and runs tests, based on a configuration file.

PIE To this end, we have developed PIE [5, 4], a DSL (domain-specific language) and runtime for precise, efficient, and expressive build programming, building forth on the Pluto [2] incremental build system.

The PIE DSL [5] is a programming language with several domain-specific concepts. Listing 1 shows an example of a build script that checks code style and runs tests. Build tasks (`taskdef`) are procedures that can invoke other tasks in their body and inspect their results. For example, `main` calls `parseYaml(./config.yaml)`, and stores its result in variable `config`. This result is then used to do *conditional building*: only if `config.checkStyle` is `true` do we require task `checkStyle`. In the `for` loop, we do *iterative building*: invoking the `runTest` task multiple times with different input `tests`. Finally, we can indicate whether we read or write to files with `requires` and `provides`.

The PIE runtime incrementally executes these build scripts based on Pluto's top-down incremental build algorithm [2]. The user requires a task (task definition + input arguments) to bring it into a consistent state and to get its result. When a task has not been executed before, PIE executes its procedure body. Otherwise, PIE only executes a task when it is deemed

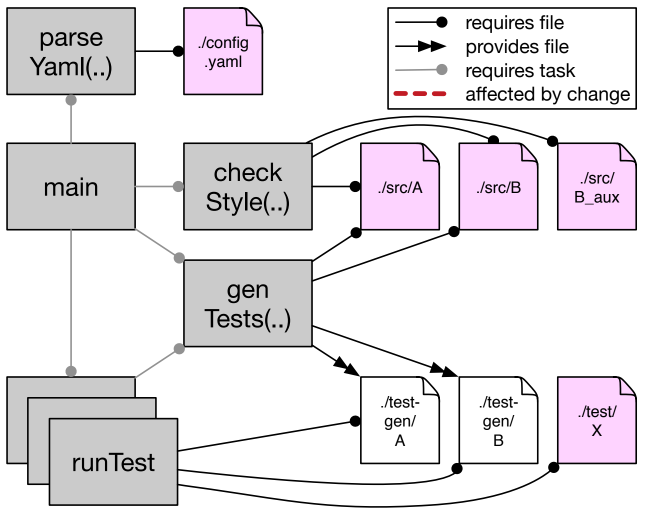

Figure 1: The initial dependency graph after executing the build script from listing 1, capturing task and file dependencies as the basis for incremental building.

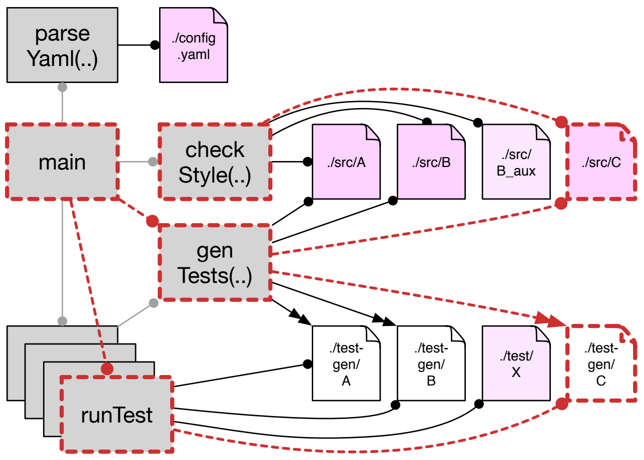

Figure 2: Dependency graph after incremental execution with a change that adds a source file. Affected tasks were executed. Notably, `main` executes a new `runTest` because `genTests` generated a new test file for `./src/C`.

inconsistent after recursively *checking* its dependencies, and only executes a task *once* each round. A task is inconsistent when any of its file dependencies have changed, or when a task it calls is inconsistent. Otherwise, the task is consistent and we can return its result from the cache. The initial dependency graph that PIE creates is shown in fig. 1, with incremental execution after a change shown in fig. 2.

A problem with the top-down build algorithm is that it needs to check the entire dependency graph to determine what has changed, which does not scale up to large dependency graphs while scaling down to many low-impact changes. To resolve this problem, we have developed a change-driven incremental build algorithm [4] that, when given a list of file changes, starts building from the bottom-up, only checking files and tasks that have been *affected by a change*, drastically speeding up rebuilds for low-impact changes.

A residual problem is that tasks which are no longer required (i.e., have no incoming task dependencies) stay in the dependency graph, and consequently keep being executed by the change-driven algorithm. To mitigate this problem, we have introduced the notion of *observability* [10]. A task is observable if it is directly or indirectly depended on by an explicitly observable task (i.e., a root task that the user deems observable). Otherwise, a task is unobserved and does not have to be checked or executed, and can even be removed from the dependency graph in a garbage collection pass.

Case Studies We have applied PIE to the interactive language development and build pipeline of the Spoofax Language Workbench [3], improving incrementality by removing under/over-approximation through precise dynamic dependencies, and improving maintainability by having only a single build script without staging instead of four separate ones [5]. We have also applied PIE to an incremental performance testing script, leading to similar benefits. Finally, PIE is being used to incrementalize the compiler of Stratego [11, 1], a term transformation meta-DSL with open extensibility.

Implementation The PIE DSL is implemented in Spoofax, and the runtime as a Java library. Both can be found online [1].

Future Work PIE currently does not support parallelism nor concurrency, as it is unclear whether a task can be parallelised or concurrently executed because of dynamic dependencies. For example, the `runTest` tasks seem to be parallelizable from glancing at fig. 1, but this dependency graph is only available *after* execution. A `runTest` tasks could start requiring other `runTest` tasks which are already concurrently executing, or other files which are being written to, causing concurrency bugs.

We have not yet investigated *distributed builds*, where build results can be cached (by uploading them, or having a server execute the build) and retrieved remotely [9]. However, since PIE has exact dependency information and uses constructive traces, it should be possible to upload/download task results. Finally, *partial evaluation* of PIE build scripts could be used to automate deployment of binaries by marking required source files of tasks as static.

Conclusion PIE is precise, as dependencies of build tasks are exactly tracked using dynamic dependencies, enabling correct and minimal incremental builds. PIE is efficient, only checking and executing tasks that have been affected by a change. Finally, PIE is expressive, as build engineers write their build scripts in a full-fledged programming language, without having to resort to workarounds or complicated design patterns.

[1]`https://github.com/metaborg/pie`

References

[1] Martin Bravenboer, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Visser. “Stratego/XT 0.17. A language and toolset for program transformation”. In: *Science of Computer Programming* 72.1-2 (2008), pp. 52–70. DOI: 10.1016/j.scico.2007.11.003.

[2] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. “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.* Ed. by Jonathan Aldrich and Patrick Eugster. ACM, 2015, pp. 89–106. ISBN: 978-1-4503-3689-5. DOI: 10.1145/2814270.2814316.

[3] Lennart C. L. Kats and Eelco Visser. “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.* Ed. by William R. Cook, Siobhán Clarke, and Martin C. Rinard. Reno/Tahoe, Nevada: ACM, 2010, pp. 444–463. ISBN: 978-1-4503-0203-6. DOI: 10.1145/1869459.1869497.

[4] Gabriël Konat, Sebastian Erdweg, and Eelco Visser. “Scalable incremental building with dynamic task dependencies”. In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018.* Ed. by Marianne Huchard, Christian Kästner, and Gordon Fraser. ACM, 2018, pp. 76–86. DOI: 10.1145/3238147.3238196.

[5] Gabriël Konat, Michael J. Steindorfer, Sebastian Erdweg, and Eelco Visser. “PIE: A Domain-Specific Language for Interactive Software Development Pipelines”. In: *Programming Journal* 2.3 (2018), p. 9. DOI: 10.22152/programming-journal.org/2018/2/9.

[6] Epperly T Kumfert G. *Software in the DOE: The Hidden Overhead of "The Build".* Tech. rep. Lawrence Livermore National Laboratory, 2002.

[7] Shane McIntosh, Bram Adams, and Ahmed E. Hassan. “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.* Ed. by Jim Whitehead and Thomas Zimmermann. IEEE, 2010, pp. 42–51. ISBN: 978-1-4244-6803-4. DOI: 10.1109/MSR.2010.5463341.

[8] Peter Miller. *Recursive make considered harmful.*

[9] Andrey Mokhov, Neil Mitchell, and Simon Peyton Jones. “Build systems a la carte”. In: *Proc International Conference on Functional Programming (ICFP’18).* ACM, Sept. 2018. URL: https://www.microsoft.com/en-us/research/publication/build-systems-la-carte/.

[10] Roelof Sol. “Observability during bottom-up execution - improving PIE’s change driven incremental build algorithm”. (to appear). MA thesis. Delft University of Technology, 2019.

[11] Eelco Visser, Zine-El-Abidine Benaissa, and Andrew P. Tolmach. “Building Program Optimizers with Rewriting Strategies”. In: *Proceedings of the third ACM SIGPLAN international conference on Functional programming.* Ed. by Matthias Felleisen, Paul Hudak, and Christian Queinnec. Baltimore, Maryland, United States: ACM, 1998, pp. 13–26. DOI: 10.1145/289423.289425.