2015 IEEE/ACM 37th IEEE International Conference on Software Engineering

Automated Decomposition of Build Targets

Mohsen Vakilian, Raluca Sauciuc, J. David Morgenthaler, Vahab Mirrokni
Google, USA
{vakilian, ralucas, jdm, mirrokni}@google.com

***Abstract*—A *(build) target* specifies the information that is needed to automatically build a software artifact. This paper focuses on *underutilized targets*—an important dependency problem that we identified at Google. An underutilized target is one with files not needed by some of its dependents. Underutilized targets result in less modular code, overly large artifacts, slow builds, and unnecessary *build and test triggers*. To mitigate these problems, programmers *decompose* underutilized targets into smaller targets. However, manually decomposing a target is tedious and error-prone. Although we prove that finding the best *target decomposition* is NP-hard, we introduce a greedy algorithm that proposes a decomposition through iterative unification of the strongly connected components of the target. Our tool found that 19,994 of 40,000 Java library targets at Google can be decomposed to at least two targets. The results show that our tool is (1) *efficient* because it analyzes a target in two minutes on average and (2) *effective* because for each of 1,010 targets, it would save at least 50% of the total execution time of the tests triggered by the target.**

I. Introduction

Software evolves rapidly [19], [25], [37]. To make the rapid evolution of software more economical and reliable, the industry has adopted Continuous Integration (CI) [11]. For each code change, a CI system first invokes the build system to build the code affected by the change. Then, it runs all the tests that transitively depend on the affected code [15], [19], [25], [37]. Google, other companies [4], [18], [22], [31], [32], and open-source projects have adopted this practice [44], [45], [49].

The faster the software evolves, the heavier the load on the CI system is. On average, the Google code repository receives over 5,500 code changes per day, which make the CI system run over 100 million test cases per day [5]. These numbers grow as Google grows. Dedicating more compute resources to the CI system is not sufficient to keep up with this growth rate. Thus, advanced technologies are needed to ensure that build and test results are delivered to programmers correctly and in a timely manner [5], [8], [15], [19], [25], [27], [37], [52].

The Google build system, like other build systems [40]–[43], [48], takes as input a set of *build files* that declare *build targets*. We refer to a *build target* as a *target* in the rest of this paper. Targets specify *what* is needed to produce an artifact such as a library or binary. A target also specifies its unique name, kind, source files, and dependencies on other targets (Figure 1). The build system decides *how* to build a given target based on the target's specification.

Build specifications capture an important architectural aspect of software, namely, the *dependency structure* between pieces of code. For example, at Google, several systems other than the build system, e.g., Integrated Development Environments (IDEs) and CI [11] systems rely on build specifications. IDEs rely on the build specifications to determine the code that needs to be indexed. Similarly, a CI system uses the build specifications to compute the set of tests affected by a code change. Despite the sophisticated caching and parallelism of Google's build system [5], [8], [15], [19], [52], slow builds, CI, and IDEs are still major issues.

Like code in languages such as C and Java, build specifications require significant, continuous maintenance. Research suggests that build maintenance accounts for 27% and 44% of code and test development, respectively [24]. Our prior work [27] showed that build specifications are prone to code smells such as *unneeded* and *missing direct dependencies*. This paper focuses on a specific code smell, which we call *underutilized targets*. An underutilized target has source files that some of its dependents do not need. Underutilized targets reduce modularity, make the builds and IDEs slower, increase the size of executables, and increase the load on the CI system by triggering unnecessary builds and tests.

A *refactoring* is a code change that preserves the behavior of the program [12], [28]. *Target decomposition*, or simply *decomposition*, is our term for a refactoring that mitigates the problems of underutilized targets. It first *decomposes* an underutilized target into smaller targets, which we refer to as *constituent targets* or simply *constituents*, and then updates the dependents of the original target to depend on only the needed constituents.

Identifying and refactoring underutilized targets is tedious and error-prone to do manually for several reasons. First, a large code base has many targets (over 40,000 targets at Google). This makes it nontrivial, if not impossible, to find the targets whose decompositions would yield the largest gains. Second, there are often many possible decompositions for a target. Choosing an effective decomposition from this large space is a daunting task. Third, manually decomposing a target is error-prone because a valid decomposition must obey the dependencies between the source files of the target. Finally, decomposing a target without updating its dependents will yield limited benefits. Once a target is decomposed into smaller, constituent targets, its dependents have to change so that they depend on the constituent targets. This refactoring is tedious and error-prone because a target can have many dependents owned by different development teams.

We propose two tools, DECOMPOSER and REFINER, for identifying and refactoring underutilized targets. DECOM-

978-1-4799-1934-5/15 $31.00 © 2015 IEEE
DOI 10.1109/ICSE.2015.34

POSER identifies underutilized targets and suggests how to decompose them to constituent targets. REFINER is a refactoring tool that updates the dependents of the underutilized targets to depend on only the needed constituent targets.

DECOMPOSER estimates the impact of a decomposition on the number of *triggers*, i.e., the number of binary and test targets that the CI system builds and runs, respectively. In addition, it suggests a decomposition using a greedy algorithm that accounts for both the *file-level* dependencies between the source files of a target and the *target-level* ones between the target and its dependents. The algorithm first computes the strongly connected components (SCCs) of the graph formed by the file-level dependencies of the target. Then, it iteratively unifies two components at a time until only two components are left. Finally, the algorithm promotes each component to a target.

Although we implemented DECOMPOSER and REFINER at Google, the underlying techniques are generalizable to other environments. These tools can be adapted to any environment that can provide its file-level and target-level dependencies. Our tools are sound assuming that the provided file-level and target-level dependencies are sound.

The results of our large-scale empirical study show that DECOMPOSER is both *efficient* and *effective* (Section X). We ran DECOMPOSER on a large, random sample of targets that consisted of 40,000 Java library targets at Google[1]. DECOMPOSER analyzes a target within minutes (mean = 2, sd = 5). Out of the 40,000 targets, DECOMPOSER found 19,994 *decomposable* targets. A decomposable target is one that has at least one *valid decomposition* (Section IV). DECOMPOSER is also effective at saving unnecessary triggers. It estimated that its proposed decompositions would significantly reduce the test execution time (minutes) per change to each target (mean = 98, sd = 1,250). On average, a decomposition proposed by DECOMPOSER reduces the total execution time of the tests triggered by the target by 12%. For each of 1,010 targets, the decompositions proposed by DECOMPOSER would save more than 50% of the execution time of the tests triggered by the target. DECOMPOSER has been deployed at Google and used by about a dozen programmers so far. This work makes several research contributions:

- We quantify the benefit of a decomposition in terms of the number of triggers that it saves (Section IV).
- We formalize the decomposition problem as a graph problem and prove that finding the best decomposition is NP-hard (Section V).
- We present the algorithm (Section VI) and implementation (Section IX) of DECOMPOSER—a tool for decomposing targets.
- We present REFINER—a tool that refactors build specifications to take advantage of a decomposition (Section VII).
- We evaluate DECOMPOSER through a large-scale empirical study in an industrial environment (Section X).

[1]For confidentiality reasons, we do not report exact statistics about the dimensions of the Google code base.

II. BUILD SYSTEM

A build system is responsible for transforming source code into libraries, executable binaries, and other artifacts. The build system takes as input a set of targets that programmers declare in build files. Figure 1 shows sample build specifications.

When a programmer issues a command to build a target, the build system first ensures that the required dependencies of the target are built. Then, it builds the desired target from its sources and dependencies. The final artifact depends on the kind of the target. For example, for Java targets, the build system produces JAR files.

A. Build Targets

Programmers have to specify four attributes in the specification of a target τ: name, kind, source files, and dependencies. The `BUILD` files shown in Figure 1 specify three targets with names `server_binary`, `server`, and `network`. $S(\tau)$ denotes the set of source files of the target named τ. The targets shown in Figure 1 set their source files to be the set of Java files in the directory that encloses the `BUILD` file. $K(\tau)$ denotes the kind of target τ, which can be `binary`, `library`, or `test`. In Figure 1, $K($`server_binary`$)$ = `binary` and $K($`network`$) = K($`server`$)$ = `library`. For both `library` and `binary` targets, the build system generates JAR files. The difference is that the JAR file for a binary target has an entry `main` method and contains all the transitive dependencies of the target. $d(\tau)$ is the set of targets that need to be built before building τ. In Figure 1, $d($`server_binary`$)$ = {`network`, `server`}.

B. Dependency Graphs

Programmers have to consider both target-level and file-level dependencies when specifying targets. The graph in Figure 2 illustrates both kinds of dependencies.

Build Graph (Target-level Dependencies). Targets specify a build graph $B = (T, E)$, where T is the set of all targets. For each $\tau_1, \tau_2 \in T$, there is an edge $(\tau_1, \tau_2) \in E$ if and only if $\tau_2 \in d(\tau_1)$.

Figure 2 shows a build graph with three library (`network`, `server`, `client`), one binary (`server_binary`), and one test (`client_tests`) targets.

The build system expects to be able to build each target after building the dependencies of the target. Thus, the build graph must be a directed acyclic graph (DAG).

The notation $u \rightsquigarrow_G v$ denotes that there is a path from vertex u to v in graph G, and $u \not\rightsquigarrow_G v$ denotes the lack thereof. For build graph B, we say that target $\tau_1 \in T$ *transitively* depends on target $\tau_2 \in T$ if and only if $\tau_1 \rightsquigarrow_B \tau_2$.

Cross References Graph (File-level Dependencies). The shape of the build graph $B = (T, E)$ is influenced by the file-level dependencies. If a source file of τ_1 references a symbol (e.g., class or method) defined in a source file of τ_2, then the build specifications must satisfy $\tau_2 \in d(\tau_1)$. More formally, let

```
java_binary(
  name = "server_binary",
  srcs = glob(["*.java"]),
  deps = [
    "network",
    "server",
  ]
)
```

(a) Contents of file `server_binary/BUILD`

```
java_library(
  name = "server",
  srcs = glob(["*.java"]),
  deps = [
    "network",
  ]
)
```

(b) Contents of file `server/BUILD`

```
java_library(
  name = "network",
  srcs = glob(["*.java"]),
  deps = []
)
```

(c) Contents of file `network/BUILD`

Fig. 1: Three `BUILD` files that declare targets `server_binary`, `server`, and `network` shown in Figure 2. Attribute `name` specifies the name of the target. The `srcs` attribute specifies the source files of the target. The expression `glob(["*.java"])` resolves to all Java files in the enclosing directory of the `BUILD` file. The `deps` attribute lists the targets that need to be built to compile the source files of the target.

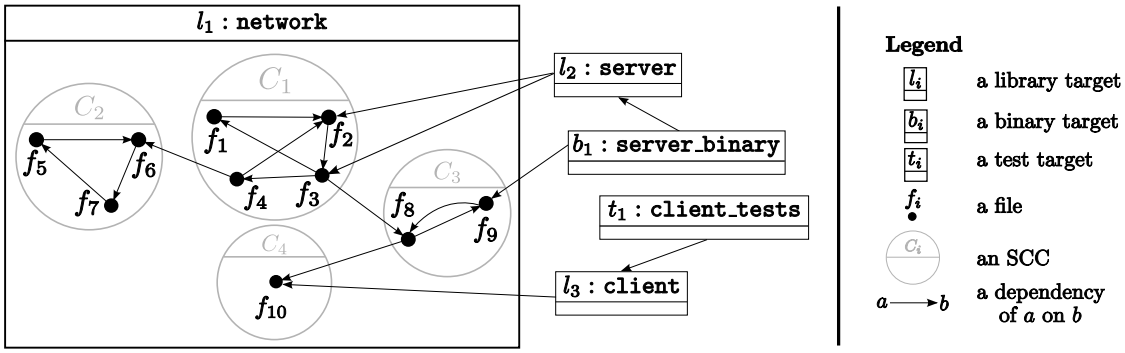

Fig. 2: A contrived graph that illustrates both target-level and file-level dependencies for an underutilized target named `network` and denoted as l_1 for brevity. C_i represents a strongly connected component (Section VI-A) of the cross references graph of l_1.

$f_1 \rightarrow f_2$ denote that file f_1 references a symbol defined in file f_2. Similarly, let $\tau_1 \rightarrow \tau_2$ denote that a file of τ_1 references a symbol defined in a file of τ_2. To simplify the discussion in the rest of the paper, we assume that $\tau_1 \rightarrow \tau_2$ if and only if $(\tau_1, \tau_2) \in E(B)$. $\tau_1 \not\rightarrow \tau_2$ indicates that $\tau_1 \rightarrow \tau_2$ does not hold.

Definition 1: The cross references between the source files of a target τ can be represented as a graph $G(\tau)$, called the cross references graph of τ. The vertices of $G(\tau)$ are members of $S(\tau)$ and there is an edge $(f_1, f_2) \in E(G(\tau))$ if and only if $f_1 \rightarrow f_2$.

The graph $G(l_1)$ is a subgraph of the graph shown in Figure 2. In this example, $G(l_1)$ consists of ten vertices corresponding to the files of l_1 and the dependency edges between these files.

C. Continuous Integration

The Google Continuous Integration (CI) system monitors every code change. The CI system computes the set of targets that may be affected by a code change. If a change affects the build graph, the CI system will update the build graph accordingly. In Figure 2, if any of the source files of `network` (i.e., $\{f_1, f_2, \cdots, f_{10}\}$) change, the CI system will invoke the build system to build the targets that transitively depend on `network`, i.e., {`server_binary`, `server`, `client`, `client_tests`} and run the tests included in the test targets that transitively depend on `network`, i.e., `client_tests`.

III. Underutilized Targets

Like ordinary source files in Java, C, and Python, build files accumulate code smells over time. A code smell specific to build files that we identified is *underutilized target*. If a target has some dependent targets that need only a subset of its source files, we consider the target underutilized. Underutilized targets lead to less modular software, larger binaries, slower builds, and unnecessary builds and tests triggered by the CI system.

Consider the example in Figure 2. Target `network` has two sets of files $S_1 = \{f_1, f_2, \ldots, f_7\}$ and $S_2 = \{f_8, f_9, f_{10}\}$. Suppose that S_1 is a set of implementation classes and S_2 is a set of interfaces and abstract classes. Files of S_1 depend on the files of S_2 but not vice versa. Target `network` is underutilized by one test target (`client_tests`). As a result, if a change affects only the files in S_1, the CI system will unnecessarily trigger the build and execution of one test target (`client_tests`). In addition, the binary created for `client_tests` will be unnecessarily large because it will include the files in S_1. As a result, an IDE will have to index unnecessary files in the transitive dependencies of `client_tests`. Underutilized targets are not specific to Google. Any build system (e.g. Make [42], Rake [48], and Gradle [43]) that allows target specifications can suffer from underutilized targets.

Dependency Granularity. The finest levels of dependencies that existing build systems (e.g., Make [42]) track are target-

level dependencies. For instance, consider a Makefile that builds a JAR file `r1.jar` from source file `j1.java` and another JAR file `r2.jar` and builds `r2.jar` from source files `j2.java` and `j3.java`. Make tracks the dependency of `r1.jar` on `r2.jar` but not on the files of `r2.jar`. As a result, if `r2.jar` changes due to a change to `j3.java`, Make will rebuild `r1.jar`, even if `r1.jar` does not depend on `j3.java`.

In theory, a CI system can save triggers by tracking dependencies at the file-level instead of target-level. However, the existing CI systems use target-level dependencies to compute the build and test triggers for three main reasons. First, maintaining the latest file-level dependencies is more expensive than that of target-level dependencies, because the number and change frequency of file-level dependencies is larger. Google has an internal, language-independent service to query file-level dependencies. However, the performance and accuracy of this service do not meet the demands of a CI system. Second, sound inference of all runtime dependencies and dependencies on data files and generated code is undecidable in general. The Google CI system avoids this problem by allowing the programmers document such dependencies of targets in build specifications. Finally, saving triggers is not the only goal of target decompositions. Even if fast and accurate tracking of file-level dependencies were possible, decomposition would have still been useful because it improves modularity.

IV. Target Decomposition

A refactoring to remove underutilized targets is to decompose them into smaller targets. We call the refactoring *target decomposition* or *decomposition* and the smaller targets *constituent* targets or simply *constituents*. For the example in Section III, this refactoring would decompose the underutilized target `network` into two constituent targets `network_a` and `network_b` such that $S(\texttt{network\_a}) = S_1$, $S(\texttt{network\_b}) = S_2$ and `network_a` depends on `network_b` (i.e., $d(\texttt{network\_a}) = \{\texttt{network\_b}\}$).

Decomposition Granularity. Intuitively, the best decomposition of a target is one that removes the largest number of unneeded dependencies from binaries and tests on the files of the target. Finer-grained decompositions can remove a larger number of unneeded dependencies. For example, decomposing a target into three constituent targets can remove more unneeded dependencies than decomposing the target into two constituent targets.

While avoiding unnecessary triggers is important, there are also other factors that influence modularity decisions. Programmers may prefer coarse-grained modules because such modules may be easier to name, may make it easier to find code, and may better match the structure of the organization. Thus, by default, DECOMPOSER proposes a decomposition of a given target into exactly two constituents. Nonetheless, DECOMPOSER can be configured to propose decompositions to more constituents.

Validity. Let $\tau/\langle\tau_1, \tau_2\rangle$ denote a decomposition of target τ into two constituent targets τ_1 and τ_2. The decomposition partitions the files of target τ between τ_1 and τ_2. It also adds two new targets τ_1 and τ_2, makes τ a target without source files, and makes τ depend on both τ_1 and τ_2.

An arbitrary partitioning of the files of a target τ into two targets may not produce a *valid decomposition*. A decomposition $\tau/\langle\tau_1, \tau_2\rangle$ is valid if and only if $\tau_2 \not\to \tau_1$. Otherwise, if $\tau_1 \to \tau_2$ and $\tau_2 \to \tau_1$, applying the decomposition will introduce a cyclic dependency between τ_1 and τ_2, which breaks the modularity of the system and is disallowed by the build system.

To simplify the exposition, we consider the decomposition $\tau/\langle\tau_1, \tau_2\rangle$ where $\tau_1 \not\to \tau_2$ and $\tau_2 \to \tau_1$ invalid, despite the fact that this decomposition keeps the build graph acyclic. We do not lose any generality by considering such a decomposition invalid, because reordering τ_1 and τ_2 produces a valid decomposition $\tau/\langle\tau_2, \tau_1\rangle$.

The decomposition $\texttt{network}/\langle\texttt{network\_a}, \texttt{network\_b}\rangle$ described above is valid because $\texttt{network\_b} \not\to \texttt{network\_a}$.

Trigger Saving. We measure the benefit of a decomposition by the number of binary and test triggers that it saves. Let $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ denote the quantitative benefit of $\tau/\langle\tau_1, \tau_2\rangle$. We refer to $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ as the *trigger saving* of $\tau/\langle\tau_1, \tau_2\rangle$.

Note that a decomposition $\tau/\langle\tau_1, \tau_2\rangle$ alone does not remove any unneeded dependencies unless the dependents of τ are changed to depend on τ_1 or τ_2. Thus, when quantifying the benefit of a decomposition, we assume that the dependents of τ will be changed to depend on τ_1 and/or τ_2 wherever possible.

Definition 2: $\mathcal{D}(\tau)$ denotes the set of binary and test targets that transitively depend on target τ.

After applying the decomposition $\texttt{network}/\langle\texttt{network\_a}, \texttt{network\_b}\rangle$, we will have $|\mathcal{D}(\texttt{network\_a})| = 1$, $|\mathcal{D}(\texttt{network\_b})| = 2$, $|\mathcal{D}(\texttt{network\_a}) - \mathcal{D}(\texttt{network\_b})| = 0$, and $|\mathcal{D}(\texttt{network\_b}) - \mathcal{D}(\texttt{network\_a})| = 1$. Note that because $\texttt{network\_a} \to \texttt{network\_b}$, we have $\mathcal{D}(\texttt{network\_a}) \subseteq \mathcal{D}(\texttt{network\_b})$. If a code change affects only the files in $S(\texttt{network\_a})$, the decomposition will save $|\mathcal{D}(\texttt{network\_b}) - \mathcal{D}(\texttt{network\_a})|$ triggers. Similarly, if a code change affects only the files in $S(\texttt{network\_b})$, the decomposition will save $|\mathcal{D}(\texttt{network\_a}) - \mathcal{D}(\texttt{network\_b})|$ triggers.

Let p_1 be the probability that a change affects only a file in $S(\tau_1)$. Similarly, let p_2 be the probability that a change affects only a file in $S(\tau_2)$. We approximate p_1 by $|S(\tau_1)|/(|S(\tau_1)| + |S(\tau_2)|)$ and p_2 by $|S(\tau_2)|/(|S(\tau_1)| + |S(\tau_2)|)$. These formula are approximations and not exact values of p_1 and p_2 because an accurate computation has to account for any change to the transitive dependencies of τ_1 and τ_2. We approximate p_1 and p_2 because their accurate computations are expensive.

Definition 3: $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$, the trigger saving of decomposition $\tau/\langle\tau_1, \tau_2\rangle$, is:

$$p_1|\mathcal{D}(\tau_2) - \mathcal{D}(\tau_1)| + p_2|\mathcal{D}(\tau_1) - \mathcal{D}(\tau_2)|,$$

where

$$p_1 = \frac{|S(\tau_1)|}{|S(\tau_1)| + |S(\tau_2)|}, \quad p_2 = \frac{|S(\tau_2)|}{|S(\tau_1)| + |S(\tau_2)|}.$$

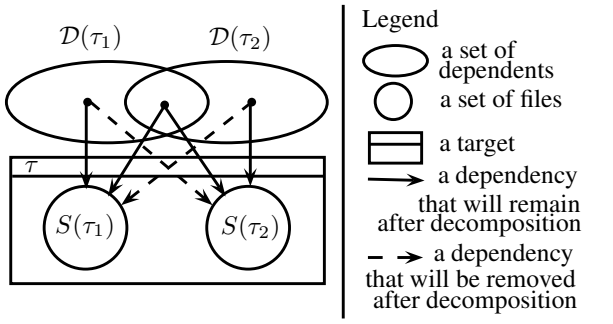

Fig. 3: Decomposition $\tau/\langle\tau_1, \tau_2\rangle$ removes unneeded dependencies (dashed arrows) that cause unnecessary build or test triggers. $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ is the average number of triggers that the decomposition would save every time a change affects the files in only $S(\tau_1)$ or only $S(\tau_2)$.

Intuitively, $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ is the expected number of binary and test targets that won't be triggered after applying the decomposition and updating the dependents of τ. The greater $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ is, the more triggers will be saved by the decomposition. Figure 3 illustrates what $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ measures.

For the decomposition `network`/$\langle$`network_a`,-`network_b`$\rangle$, we have $p_1 = \frac{7}{10}$ and $p_2 = \frac{3}{10}$. Thus, $\Delta($`network`/$\langle$`network_a`, `network_b`$\rangle) = \frac{7}{10}\cdot 1 + \frac{3}{10}\cdot 0 = 0.7$. This implies that decomposing target `network` can save on average 0.7 triggers every time a change affects only $S($`network_a`$)$ or only $S($`network_b`$)$. Although the saving is small in this contrived example, decomposing targets yields significant benefits in practice (Section X).

V. Hardness of Decomposition

Theorem 1: Given a target τ, finding the decomposition $\tau/\langle\tau_1, \tau_2\rangle$ that maximizes $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ is an NP-hard problem.

Proof: We prove NP-hardness by showing a reduction from the maximum clique problem in graph theory. The proof is included in an accompanying technical report [39]. ■

VI. Decomposition Algorithm

Since finding the best decomposition is an NP-hard problem, we propose an efficient greedy algorithm that finds effective decompositions in practice. Our algorithm suggests a decomposition in the following steps:

1) Compute the strongly connected components (SCCs) of the cross references graph of the given target.
2) Find the binary and test targets that transitively depend on each SCC.
3) Partition the SCCs of the target into two sets with a goal of maximizing the trigger saving (Definition 3).
4) Update the build specifications to apply the decomposition.

The rest of this section describes the above steps.

A. Strongly Connected Components (SCCs)

A directed graph G is *strongly connected* if and only if for each pair of vertices $v_1, v_2 \in V(G)$, $v_1 \rightsquigarrow_G v_2$ and $v_2 \rightsquigarrow_G v_1$. A strongly connected component of a graph G is a maximal subgraph of G that is strongly connected. We refer to a strongly connected component as an SCC. A *component* is a subgraph that is either an SCC or the union of two components.

$S(\tau, C)$ denotes the set of files of target τ in component C. For example, target `network` in Figure 2 consists of four SCCs. We have $S(\tau, C_1) = \{f_1, f_2, f_3, f_4\}$.

The SCCs of $G(\tau)$ form the smallest units of decomposing target τ. That is, any valid decomposition must assign all files of an SCC to the same constituent target. Otherwise, there will be a cyclic dependency between the constituent targets. Thus, the decomposition problem reduces to decomposing the set of SCCs instead of the set of files.

Condensation Graph. If each SCC of G is contracted to a single vertex, the resulting graph is the *condensation graph* of G denoted as $\mathcal{C}(G)$. In Figure 2, $\mathcal{C}(G(l_1))$ has four vertices C_1, C_2, C_3, and C_4 and three edges. As a starting point, our algorithm computes $\mathcal{C}(G(\tau))$ using a standard DFS-based algorithm [10] that runs in $O(N)$ time and space, where $N = |V(G(\tau))|$.

If there is no limit on the number of constituent targets and $\mathcal{C}(G(\tau))$ has n vertices corresponding to SCCs (C_1, C_2, $\cdots$, C_n), then the best decomposition of τ will be $\tau/\langle\tau_1, \tau_2, \cdots, \tau_n\rangle$, where $S(\tau_i) = S(\tau, C_i)$ for each $i \in \{1, \ldots, n\}$. However, due to the potential drawbacks of such a fine-grained decomposition (Section IV), our algorithm proposes a decomposition to only two constituent targets by default.

B. Dependents

A decomposition $\tau/\langle\tau_1, \tau_2\rangle$ is ideal if it maximizes $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ (Definition 3). $\Delta(\tau/\langle\tau_1, \tau_2\rangle)$ depends on $\mathcal{D}(\tau_1)$ and $\mathcal{D}(\tau_2)$ (Definition 2), i.e., the set of binary and test targets that transitively depend on τ_1 and τ_2, respectively. To find constituent targets τ_1 and τ_2, our algorithm first computes $\mathcal{D}(\tau, C)$ for each SCC c. $\mathcal{D}(\tau, C)$ is the set of binary and test targets that transitively depend on SCC C of $G(\tau)$. In Figure 2, $\mathcal{D}($`network`$, C_i)$ is a set of b_j and t_k targets that can reach a file in C_i by following the dependency edges. In Figure 2, we have $\mathcal{D}($`network`$, C_1) = \mathcal{D}($`network`$, C_2) = \mathcal{D}($`network`$, C_3) = \{$`server_binary`$\}$ and $\mathcal{D}($`network`$, C_4) = \{$`server_binary`, `client_tests`$\}$. Finally, we compute $\mathcal{D}(\tau)$, the set of binary and test targets that transitively depend on τ by taking the union of $\mathcal{D}(\tau, C)$ for all SCC C of $G(\tau)$.

C. Unifying Components

We define *unification* as an operation that takes two components C_1 and C_2 of $G(\tau)$ and creates a new component C such that $S(\tau, C) = S(\tau, C_1) \cup S(\tau, C_2)$ and contracts the two vertices of $\mathcal{C}(G(\tau))$ corresponding to C_1 and C_2 to a vertex corresponding to C. If C_1 and C_2 are unified to C, we will have $\mathcal{D}(\tau, C) = \mathcal{D}(\tau, C_1) \cup \mathcal{D}(\tau, C_2)$.

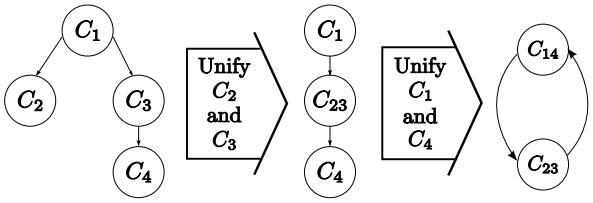

Fig. 4: Unifying the components of the cross references graph of target `network` in Figure 2. The graph on the left is $\mathcal{C}(G(\texttt{network}))$. First, C_2 and C_3 are unified to C_{23}. Then, C_1 and C_4 are unified to C_{14}. The final condensation graph (on the right) is invalid because it has a cycle. As a result, a decomposition corresponding to C_{14} and C_{23} is invalid, too.

Figure 4 shows two subsequent unifications applied on the condensation graph of target l_1 in Figure 2. The first unification contracts vertices C_2 and C_3 to a new vertex C_{23}, where $S(l_1, C_{23}) = S(l_1, C_2) \cup S(l_1, C_3) = \{f_5, f_6, f_7, f_8, f_9\}$.

1) Iterative Unification: After computing the SCCs of the cross references graph of a target, the algorithm iteratively unifies two components at each step until only two are left. The two remaining components form the two new constituent targets. Unification does not increase the trigger saving. Following a greedy scheme, at each step, the algorithm unifies two components whose unification incurs the least cost. Let $\delta(\tau, C_1, C_2)$ be the cost of unifying components C_1 and C_2 of $G(\tau)$. Intuitively, $\delta(\tau, C_1, C_2)$ is the average number of triggers per change that would be saved if C_1 and C_2 are not unified. Similar to Definition 3, we define $\delta(\tau, C_1, C_2)$ as

$$p_1|\mathcal{D}(\tau, C_2) - \mathcal{D}(\tau, C_1)| + p_2|\mathcal{D}(\tau, C_1) - \mathcal{D}(\tau, C_2)|,$$

where

$$p_1 = \frac{|S(\tau, C_1)|}{|S(\tau, C_1)| + |S(\tau, C_2)|}, \quad p_2 = \frac{|S(\tau, C_2)|}{|S(\tau, C_1)| + |S(\tau, C_2)|}.$$

At each step, the greedy algorithm unifies two components C_1 and C_2 such that $\delta(\tau, C_1, C_2) = \min_{i,j} \delta(\tau, C_i, C_j)$. For target l_1 in Figure 2, the algorithm first unifies C_1 and C_2 to C_{12}, because it incurs the least cost ($\delta(l_1, C_1, C_2) = 0$). Next, it unifies C_3 and C_4 to C_{34}, because it has the smallest unification cost ($\delta(l_1, C_3, C_4) = \frac{2}{3}$). Finally, it will turn C_{12} and C_{34} into constituent targets.

2) Avoiding Invalid Decompositions: The unification algorithm as described above may produce invalid decompositions. Consider the example condensation graph in Figure 4. Suppose the greedy algorithm first unifies C_2 and C_3 into C_{23}, and then C_1 and C_4 into C_{14}. These unifications produce an invalid decomposition. This is because the targets corresponding to C_{23} and C_{14} introduce a circular dependency to the build graph.

Lemma 1: Contracting two vertices that are adjacent in a topological ordering of a DAG results in another DAG.

Lemma 2: Contracting two root vertices (i.e., vertices without incoming edges) or two leave vertices (i.e., vertices without outgoing edges) of a DAG results in another DAG.

```
   input  : B, the build graph
   input  : τ, an underutilized target
   input  : τ1, τ2, constituent targets of τ (τ1 ∉ Deps(τ2))
1  foreach u ∈ V(B) where (u, τ) ∈ E(B) do
2      E(B) ← E(B) - (u, τ)
3      if not builds(u) then
4          E(B) ← E(B) ∪ (u, τ2)
5          if not builds(u) then
6              E(B) ← E(B) - (u, τ2) ∪ (u, τ1)
7              if not builds(u) then
8                  E(B) ← E(B) - (u, τ1) ∪ (u, τ)
```

Fig. 5: Given an underutilized target τ, REFINER generates a patch for each dependent of τ that does not need to depend on both constituents of τ.

We use Lemmas 1 and 2 to guarantee that unifying components always produces a valid decomposition. Rather than considering the unifications of all pairs of components, we make the algorithm consider the unifications of only those pairs of components that are both roots, both leaves, or adjacent in a topological ordering of the condensation graph.

D. Constituent Targets

Currently, rewriting the build specifications to introduce the constituent targets is semi-automated. The iterative unification of the components of τ terminates when only two components are left. Next, the programmer has to set $S(\tau)$ to $\emptyset$ and specify the constituent targets whose source files correspond to those of the two components. The programmer has to set $d(\tau_1)$ to $\{\tau_2\}$, $d(\tau_2)$ to $d(\tau)$, and $d(\tau)$ to $\{\tau_1, \tau_2\}$. Finally, the programmer has to run a separate tool that removes unneeded dependencies of targets and converts indirect dependencies to direct ones.

VII. DEPENDENCY REFINEMENT

Decomposing an underutilized target alone brings several benefits. First, it improves the modularity of the system. Second, it reduces the build time when a code change does not affect all the constituent targets. Third, new targets that programmers will add in future can depend only on the needed constituent targets instead of the whole underutilized target. Such finer-grained dependencies reduce the overall build time and size of binaries. Nonetheless, to unleash the full benefits of a decomposition, the dependents of the target need to change to depend on only the needed constituent targets. This change is a refactoring because it just makes the build-time dependencies finer-grained and does not affect the behavior of any program. We call this refactoring *dependency refinement*.

If the underutilized target has many dependents, the dependency refinement will become time-consuming to do manually. Thus, we developed a tool called REFINER to automate this refactoring. Given an underutilized target, REFINER automatically and safely generates a patch that refines the dependencies of the dependents of the underutilized target to only the needed constituents.

Figure 5 lists the pseudocode of REFINER. REFINER examines every dependent u of the given underutilized target τ (line 1). Let τ_1 and τ_2 be the constituents of τ such that τ_2 does

not depend on τ_1. First, REFINER removes the dependency of u on τ (line 2). If u continues to build successfully, this suggests that the dependency on u was unneeded. Otherwise, REFINER first tries a dependency on τ_2 (line 4) and then τ_1 (line 6). If u cannot be built successfully with a dependency on either τ_1 or τ_2, it means that u needs both τ_1 and τ_2. In this case, REFINER adds back the dependency on τ (line 8). While DECOMPOSER proposes a change to a single build file, REFINER often generates a patch that affects many build files.

Our prior work on enforcing direct dependencies [27] prepares the foundation for applying REFINER. The direct dependencies of a target define the symbols that the target references directly. Enforcing direct dependencies requires the programmers to explicitly specify these dependencies even if they are already in the transitive dependencies of the target. Enforcing the specification of direct dependencies allows REFINER to focus on only the direct dependents of the underutilized target.

Depending on the number of dependents of the underutilized target, REFINER may take several hours to run. The bottleneck is in building all the dependents affected by the decomposition. We run all the tests that are affected by the patch that REFINER generates. If REFINER does not cause new breakages, we submit the patch to be reviewed by the programmers that own the build specifications affected by the patch. Depending on the number and availability of the owners, the review process may take from several days to weeks.

VIII. SOUNDNESS

We say that the target-level and file-level dependency graphs are sound if and only if all the dependencies that appear in source and build files are included in these graphs.

Soundness of DECOMPOSER. We show that if the file-level and target-level dependency graphs are sound, the greedy decomposition algorithm will also be sound. That is, applying the resulting decomposition does not cause the build graph to become cyclic or miss a dependency that exists in the source or build files.

The decomposition $\tau/\langle\tau_1, \tau_2\rangle$ does not affect the target-level dependencies that do not involve τ, τ_1, and τ_2. So, we only need to show that the decomposition updates the dependencies involving τ, τ_1, and τ_2 in a sound way.

Lemma 1 implies that unifying only the neighboring components of the cross references graph prevents the decomposition from adding any cycles to the build graph.

The decomposition distributes the original dependencies of τ between τ_1 and τ_2 and makes τ depend on τ_1 and τ_2. Those targets that used to depend on τ do not miss any dependency either, because the transitive dependencies of τ before and after the decomposition are the same. Thus, the decomposition does not miss any dependencies.

Soundness of REFINER. If the target-level dependencies are sound, we show that REFINER is also sound. REFINER may change only the dependencies of each dependent u of the underutilized target τ. REFINER relies on the build system to ensure that the changes to target-level dependencies do not break the build of u. Because each target has to specify its direct dependencies, dependents of u will continue to build after the changes to u.

IX. IMPLEMENTATION

DECOMPOSER is a Java program that leverages several internal Google services through Remote Procedure Calls. It uses Google Protocol Buffers [47] to exchange data with these services. DECOMPOSER gets the file-level dependencies of a target from a service. It uses these dependencies to construct the cross references graph (e.g., $G(\texttt{network})$ in Figure 2) and compute the dependencies of other targets on the files of the target under analysis (e.g., dependency edges $(\texttt{server}, f_2)$, $(\texttt{server\_binary}, f_9)$, and $(\texttt{client}, f_8)$ in Figure 2). For target-level dependencies, DECOMPOSER uses an in-memory graph [15] that the CI system maintains to compute the targets affected by a change. DECOMPOSER queries a database that contains the log data of the CI system to estimate the trigger savings in terms of past test execution times. To make the implementation more reusable and extensible to open repositories (e.g., the Maven Central Repository [46]), we employed the Facade design pattern [13, pp. 185–193] to provide abstractions for the services that DECOMPOSER relies on.

DECOMPOSER uses FlumeJava [9] for analyzing targets in parallel. FlumeJava is a Java framework developed at Google for MapReduce computations. When run in parallel mode, DECOMPOSER distributes the input list of targets among thousands of FlumeJava mappers that run independently of each other in Google's data centers.

REFINER is a Python program that relies on the build system, the target-level dependencies, and a headless tool for rewriting build specifications.

X. EMPIRICAL RESULTS

We evaluated DECOMPOSER to answer the following research questions:

- **RQ$_1$:** What percentage of targets can be decomposed?
- **RQ$_2$:** How effective are the decompositions that DECOMPOSER suggests?
- **RQ$_3$:** How efficient is DECOMPOSER?
- **RQ$_4$:** How receptive are programmers to the changes that DECOMPOSER and REFINER propose?

A. RQ$_1$: What percentage of targets can be decomposed?

We ran DECOMPOSER on a random sample of targets at Google comprising of 40,000 Java library targets. DECOMPOSER reported that 19,994 (50%) of the analyzed targets were *decomposable*. A target is decomposable if and only if its cross references graph has at least two SCCs. DECOMPOSER found that decomposable targets have on average ten files, nine SCCs, and 2,062 dependents (Table I).

TABLE I: STATISTICS ABOUT DECOMPOSABLE TARGETS AS ESTIMATED BY DECOMPOSER. "TRIGGER TIME" IS THE TOTAL EXECUTION TIME OF THE TESTS THAT A CHANGE TO A TARGET TRIGGERS. "SAVED TRIGGERS" IS COMPUTED ACCORDING TO DEFINITION 3. "SAVED TRIGGERS PCT." IS THE RATIO OF "SAVED TRIGGERS" OVER "DEPENDENTS". "SAVED TRIGGER TIME" IS THE TOTAL TEST EXECUTION TIME OF THE SAVED TRIGGERS. "SAVED TRIGGER TIME PCT." IS THE RATIO OF "SAVED TRIGGER TIME" OVER "TRIGGER TIME". "DECOMPOSER EXEC. TIME" IS THE EXECUTION TIME OF DECOMPOSER ITSELF.

	Min	Max	Mean	SD
Files	2	1,098	10	27
SCCs	2	903	9	22
Dependents	0	674,992	2,062	24,234
Trigger Time (mins)	0	127,860	845	5,978
Saved Triggers (Δ)	0	396,360	276	6,245
Saved Triggers Pct. (Δ%)	0	99	11	19
Saved Trigger Time (mins)	0	60,837	98	1,250
Saved Trigger Time Pct.	0	99	12	22
DECOMPOSER Exec. Time (mins)	1	369	2	5

B. RQ_2: How effective are the decompositions that Decomposer suggests?

We measure the effectiveness of a decomposition by calculating the number ($RQ_{2.1}$) and percentage ($RQ_{2.2}$) of saved triggers and the duration ($RQ_{2.3}$) and percentage ($RQ_{2.4}$) of saved test execution time.

Tables II–V demonstrate the effectiveness of DECOMPOSER. The first column of each of these tables partitions the values of a metric into multiple intervals. The second and third columns report the number and percentage of the targets that fall within each interval, respectively. The fourth and fifth columns are cumulative versions of the second and third columns, respectively. The distributions consistently show that decomposing a small fraction of targets yields substantial benefits. By estimating the benefits of decomposing each target, DECOMPOSER enables the programmers to focus on the decompositions with the largest gains.

TABLE II: DISTRIBUTION OF THE NUMBER OF SAVED TRIGGERS

Saved Triggers	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
$[900, \infty)$	355	6.9	355	6.9
$[800, 900)$	29	0.6	384	7.5
$[700, 800)$	26	0.5	410	8.0
$[600, 700)$	36	0.7	446	8.7
$[500, 600)$	60	1.2	506	9.9
$[400, 500)$	72	1.4	578	11.3
$[300, 400)$	101	2.0	679	13.2
$[200, 300)$	184	3.6	863	16.8
$[100, 200)$	322	6.3	1,185	23.1
$(0, 100)$	3,944	76.9	5,129	100.0

1) $RQ_{2.1}$: How many triggers can Decomposer save?: DECOMPOSER estimates that the decompositions it suggests for 26% of the decomposable targets (5,129 of 19,994) would save at least one trigger. Moreover, it found that on average decomposing a target saves 276 triggers (Table I) per change to the target. Table II shows that decomposing any one of 355 targets would save at least 900 triggers of the target.

TABLE III: DISTRIBUTION OF THE PERCENTAGE OF SAVED TRIGGERS

Saved Triggers (%)	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
$[90, 100]$	31	0.6	31	0.6
$[80, 90)$	71	1.4	102	2.0
$[70, 80)$	124	2.4	226	4.4
$[60, 70)$	248	4.8	474	9.2
$[50, 60)$	533	10.4	1,007	19.6
$[40, 50)$	632	12.3	1,639	32.0
$[30, 40)$	618	12.0	2,257	44.0
$[20, 30)$	629	12.3	2,886	56.3
$[10, 20)$	707	13.8	3,593	70.1
$(0, 10)$	1,536	29.9	5,129	100.0

TABLE IV: DISTRIBUTION OF SAVED TRIGGER TIMES

Saved Trigger Time (min)	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
$[60, \infty)$	1,145	25.1	1,145	25.1
$[30, 60)$	287	6.3	1,432	31.3
$[10, 30)$	633	13.9	2,065	45.2
$[5, 10)$	442	9.7	2,507	54.9
$[2, 5)$	641	14.0	3,148	68.9
$[1, 2)$	521	11.4	3,669	80.3
$(0, 1)$	900	19.7	4,569	100.0

2) $RQ_{2.2}$: What percentage of triggers can Decomposer save?: The decompositions suggested by DECOMPOSER save 11% of the triggers on average (Table I). Table III shows that decomposing any one of only 31 targets would save at least 90% of the triggers per change to the target.

3) $RQ_{2.3}$: How much test execution time can Decomposer save?: The decompositions that DECOMPOSER suggests save 98 minutes of the test execution time of a decomposable target

TABLE V: DISTRIBUTION OF THE PERCENTAGE OF SAVED TRIGGER TIME

Saved Triggers Time (%)	Freq.	Freq. (%)	Cum. Freq.	Cum. Freq. (%)
$[90, 100]$	62	1.4	62	1.4
$[80, 90)$	87	1.9	149	3.3
$[70, 80)$	153	3.3	302	6.6
$[60, 70)$	246	5.4	548	12.0
$[50, 60)$	462	10.1	1,010	22.1
$[40, 50)$	601	13.2	1,611	35.3
$[30, 40)$	492	10.8	2,103	46.0
$[20, 30)$	448	9.8	2,551	55.8
$[10, 20)$	533	11.7	3,084	67.5
$(0, 10)$	1,485	32.5	4,569	100.0

TABLE VI: THE RATIO OF THE DURATION OF EACH PHASE OF DECOMPOSER OVER THE EXECUTION TIME OF DECOMPOSER AVERAGED OVER ALL OF THE 40,000 ANALYZED TARGETS.

Phase	Duration Pct.
Constructing the cross references graph	4
Computing the SCCs	0
Computing the target-level dependencies	66
Computing the dependents of SCCs	30
Unifying SCCs	0

on average (Table I). DECOMPOSER estimates the execution time of the saved test triggers by computing the average execution time of the saved test targets during the past day. Table IV indicates that decomposing any of 1,145 targets would reduce the test execution time per change to the target by at least an hour.

4) $RQ_{2.4}$: What percentage of test execution time can Decomposer save?: On average, a decomposition that DECOMPOSER proposes for a target would save 12% of the execution time of the tests that are triggered by a change to the target (Table I). This number is close to the average percentage of triggers that are saved by a decomposition (Section X-B2). This is not surprising because saving more triggers tends to save more test execution time. Table V indicates that the decompositions proposed by DECOMPOSER for 1,010 decomposable targets would save at least 50% of the test execution time of each of these targets.

C. RQ_3: How efficient is Decomposer?

On average, DECOMPOSER analyzes a target in two minutes (Table I). This implies that if we had run DECOMPOSER on the 40,000 targets sequentially, it would have taken it more than 55 days to finish. DECOMPOSER analyzes all these targets in parallel overnight. Table VI shows the average breakdown of the execution time of each phase of DECOMPOSER. The table shows that the most expensive phases of the algorithm are computing the target-level dependencies and the dependents of SCCs. The target-level dependencies are represented as a large directed graph. Each edge of this graph indicates a dependency of a target on another target. Deserializing this graph from the file system is expensive. Computing the dependents of SCCs is expensive for targets with many dependents.

D. RQ_4: How receptive are programmers to the changes that Decomposer and Refiner propose?

As a preliminary evaluation, we selected seven targets for decomposition. DECOMPOSER estimated high trigger savings for these targets and the dependents of these targets declared all their direct dependencies. Every code change at Google gets peer reviewed. We submitted code changes based on the results of DECOMPOSER for these seven targets. Six code changes got reviewed, four of which got approved. Two code changes got rejected, because the reviewer expected the target to change rarely. This experience highlights an opportunity to improve DECOMPOSER (Section XII). We submitted two code changes created by REFINER, both of which got approved. The number of reviewed code changes is low, because the review process is slow and can take up to several weeks, especially when the changes affect code owned by multiple teams or the owners are not available. Nonetheless, the preliminary results suggest that programmers are receptive to the code changes generated by DECOMPOSER and REFINER.

XI. RELATED WORK

Despite the recent move of the software industry to CI [4], [19], [22], [31], [32], there has been little research on CI. The rest of this section overviews several empirical studies, code smell detection and refactoring tools for build specifications and discusses our work with respect to software remodularization and regression testing.

Empirical Studies. McIntosh *et al.* [24] studied the version histories of ten projects and found that build maintenance accounts for up to 27% overhead on source code development and 44% overhead on test development. In another study of six open-source projects [23], McIntosh *et al.* found that the size of build files and source files are highly correlated. In short, these studies show that build maintenance incurs significant engineering cost. This cost calls for tool support for evolving build specifications.

Underutilized Targets. Build Analyzer is an interactive commercial tool for optimizing the build time of C/C++ code [36]. It allows programmers to identify *fat headers*, the header files that are build bottlenecks, and decompose them into two smaller header files. Little has been reported about the decomposition algorithm and empirical evaluation of Build Analyzer. Although Build Analyzer refactors header files and not build specifications, fat headers and underutilized targets are related code smells.

In our prior work [27], we discussed several code smells specific to build specifications, including under-declared dependencies, zombie targets, and visibility debt. We introduced a tool called Clipper that takes a binary target as input and ranks the libraries in the transitive closure of the dependencies of the binary by their utilization rates, i.e., the percentage of the symbols of the library that are used by the binary. Clipper helps programmers find the libraries that are bringing too many unneeded symbols to the binary. Clipper, DECOMPOSER, and REFINER are complementary tools. Programmers can use Clipper to find underutilized targets and then use DECOMPOSER and REFINER to decompose them.

Software Remodularization. Remodularization is decomposing a code base that is almost monolithic into modules [50]. Researchers have developed tools for remodularizing legacy software. These tools employ clustering [3], [21], [38], [51], search-based [6], [26], [30], or information retrieval [20] techniques to find a set of modules that optimizes some metrics. These metrics are usually inspired by properties such as high cohesion and low coupling [1], [7]. While existing remodularization tools target legacy software with poor modularity, our tools are intended for modern software that is relatively modular but can benefit from finer-grained modules.

Analyzing, Visualizing, and Refactoring Makefiles. MAKAO [2] is a tool that visualizes Makefiles by analyzing their dynamic build traces. It also supports refactorings such as target creation. SYMake [35] is a static analysis tool that can detect several code smells of Makefiles such as cyclic dependencies and duplicate prerequisites and supports refactorings such as target creation and renaming. While MAKAO and SYMake support basic refactorings of Makefiles, neither can detect or refactor underutilized targets.

Test Selection. The goal of *test-selection techniques* [14], [16], [17], [29], [33], [34], [53] is to select a subset of the tests of one version of a program to run on a future version of the program without compromising the fault-detection capability of the test suite.

Since we defined the effectiveness of a decomposition in terms of the triggers that it saves (Definition 3), decomposing underutilized targets can be viewed as a test-selection technique. Target decomposition is a refactoring that makes the test-selection technique of the CI system more effective. However, the benefits of decomposing targets are not limited to test selection. Decomposing underutilized targets can reduce build time, binary size, and improve the modularity of code and the performance of IDEs (Section III).

XII. Limitations and Future Work

Generalizability. The evaluation results are limited to Java targets at Google. Nonetheless, DECOMPOSER and REFINER are both language independent, because they operate at the level of files and targets. We have designed DECOMPOSER and REFINER for adaptability to software repositories outside Google, e.g., the Maven Central Repository [46]. The target-level dependency graph of the Maven Central Repository can be constructed from the POM files. Similar to Google build specifications, the POM files specify the targets and their dependencies. The file-level dependency graph can be extracted from the cross references within and between the artifacts (e.g., JAR files) published in the Maven Central Repository. Once the target-level and file-level dependency graphs are computed, DECOMPOSER and REFINER can use these graphs to decompose targets and refine dependencies, respectively. Sometimes, Google programmers manually decompose underutilized JAR files built from open-source code. This anecdote indicates the practical value of automated decomposition of open-source targets.

Soundness. DECOMPOSER and REFINER are sound as long as the target-level and file-level dependency graphs are sound (Section VIII). Currently, the target-level dependencies miss the dependencies on generated targets, and the file-level dependencies include only the static dependencies. As the services that report these dependencies become more accurate, DECOMPOSER and REFINER benefit, too.

Objective Function. DECOMPOSER uses the number of saved triggers as an objective function to find a decomposition. In future, we plan to experiment with different objective functions. Alternative objective functions can optimize the decomposition to reduce the size of binaries. In addition, the objective function can be extended to take the *change rates* of files into account. Files that rarely change trigger few tests. Finally, future research can explore the impact of *code co-evolution* on decomposition. For instance, decomposing a target into two constituents that are often affected by the same changes will save few triggers.

Decomposition Algorithm. DECOMPOSER employs a greedy algorithm to suggest a decomposition. This algorithm is fast and can suggest decompositions to an arbitrary number of constituents. However, finding an approximation algorithm with a provable guarantee of closeness to the optimal decomposition or proving the lack of such an algorithm are open problems. Future research can study alternative decomposition algorithms.

Adoption. So far, about a dozen programmers at Google have used DECOMPOSER. Our vision is to integrate DECOMPOSER into the programming workflow to gain a wider adoption. Ideally, DECOMPOSER would continuously monitor every code change and suggest that programmers decompose a target whenever the benefit of the decomposition goes above a certain threshold.

XIII. Conclusions

Build specifications embody the *dependency structure* of large-scale software. *Build specifications are code, too.* Like any other code, build specifications accumulate *code smells* as software *evolves*. This paper focuses on a specific code smell of build specifications that we identified in Google's code base, namely, *underutilized build targets*. We present a tool for *large-scale identification* and *decomposition* of underutilized build targets. Our evaluation results show that our tool is both *effective* and *efficient* at (1) estimating the benefits of decomposing build targets, and (2) proposing decompositions of build targets. Besides the promising results of our tool at Google, perhaps a broader contribution of our work is highlighting a challenging problem that the software industry faces: *improving the quality of build specifications at scale*.

Acknowledgments

The first author was employed by Google while working on this project. We thank Nicholas Chen, Munawar Hafiz, Ralph Johnson, Darko Marinov, Stas Negara, Tao Xie, and the student participants of the software engineering seminar at Illinois for their comments on a draft of this paper. We also thank Eddie Aftandilian, John Penix, Sanjay Bhansali, Kevin Bourrillion, Robert Bowdidge, Dana Dahlstrom, Misha Gridnev, Jeremy Manson, John Micco, Ben St. John, Jeffrey van Gogh, Collin Winter, and many others at Google for their suggestions and engineering support.

References

[1] H. Abdeen, H. Sahraoui, O. Shata, N. Anquetil, and S. Ducasse. Towards Automatically Improving Package Structure while Respecting Original Design Decisions. In *Proceedings of the 20th Working Conference on Reverse Engineering (WCRE)*, pages 212–221, 2013.

[2] B. Adams, H. Tromp, K. De Schutter, and W. De Meuter. Design Recovery and Maintenance of Build Systems. In *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM)*, pages 114–123, 2007.

[3] N. Anquetil and T. C. Lethbridge. Experiments with Clustering as a Software Remodularization Method. In *Proceedings of the 6th Working Conference on Reverse Engineering (WCRE)*, pages 235–255, 1999.

[4] C. AtLee, L. Blakk, J. O'Duinn, and A. Z. Gasparnian. Firefox Release Engineering. In *The Architecture of Open Source Applications*, volume 2. Lulu, 2012.

[5] M. Barnathan, G. Estren, and P. Lebeck-Jobe. Building Software at Google Scale. http://www.youtube.com/watch?v=2qv3fcXW1mg, 2012.

[6] G. Bavota, F. Carnevale, A. D. Lucia, M. D. Penta, and R. Oliveto. Putting the Developer in-the-Loop: An Interactive GA for Software Remodularization. In *Proceedings of the 4th International Symposium on Search Based Software Engineering (SSBSE)*, pages 75–89, 2012.

[7] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto. Software Re-Modularization Based on Structural and Semantic Metrics. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 195–204, 2010.

[8] M. Besta, Y. Miretskiy, and J. Cox. Build in the Cloud: Distributing Build Outputs. [Blog post] http://goo.gl/jaQTiF, 2011.

[9] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: Easy, Efficient Data-Parallel Pipelines. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 363–375, 2010.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Elementary Graph Algorithms. In *Introduction to Algorithms*. The MIT Press, 2009.

[11] P. M. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.

[12] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[13] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[14] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An Empirical Study of Regression Test Selection Techniques. *ACM Transactions on Software Engineering and Methodology*, 10:184–208, 2001.

[15] P. Gupta, M. Ivey, and J. Penix. Testing at the Speed and Scale of Google, 2011. [Blog post] http://goo.gl/dmOUMN.

[16] M. J. Harrold, J. A. Jones, T. Li, D. Liang, and A. Gujarathi. Regression Test Selection for Java Software. In *Proceedings of the 2001 ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 312–326, 2001.

[17] M. J. Harrold and M. L. Souffa. An Incremental Approach to Unit Testing during Maintenance. In *Proceedings of the Conference on Software Maintenance (ICSM)*, pages 362–367, 1988.

[18] C. H. P. Kim, D. Marinov, S. Khurshid, D. Batory, S. Souto, P. Barros, and M. d'Amorim. SPLat: Lightweight Dynamic Analysis for Reducing Combinatorics in Testing Configurable Systems. In *Proceedings of the ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, pages 257–267, 2013.

[19] A. Kumar. Development at the Speed and Scale of Google. QCon San Francisco, http://goo.gl/hCPQxZ, 2010.

[20] J. I. Maletic and A. Marcus. Supporting Program Comprehension Using Semantic and Structural Information. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, pages 103–112, 2001.

[21] O. Maqbool and H. Babri. Hierarchical Clustering for Software Architecture Recovery. *IEEE Transactions on Software Engineering*, pages 759–780, 2007.

[22] D. Marsh. From Code to Monkeys: Continuous Delivery at Netflix. QCon San Francisco, http://goo.gl/lQWQrY, 2013.

[23] S. McIntosh, B. Adams, and A. E. Hassan. The Evolution of Java Build Systems. *Empirical Software Engineering*, pages 578–608, 2012.

[24] S. McIntosh, B. Adams, T. H. Nguyen, Y. Kamei, and A. E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, pages 141–150, 2011.

[25] J. Micco. Tools for Continuous Integration at Google Scale. http://www.youtube.com/watch?v=KH2_sB1A6lA, 2012.

[26] B. S. Mitchell and S. Mancoridis. On the Automatic Modularization of Software Systems Using the Bunch Tool. *IEEE Transactions on Software Engineering*, pages 193–208, 2006.

[27] J. D. Morgenthaler, M. Gridnev, R. Sauciuc, and S. Bhansali. Searching for Build Debt: Experiences Managing Technical Debt at Google. In *Proceedings of the 3rd International Workshop on Managing Technical Debt (MTD)*, pages 1–6, 2012.

[28] W. F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[29] A. Orso, N. Shi, and M. J. Harrold. Scaling Regression Testing to Large Software Systems. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 241–251, 2004.

[30] K. Praditwong, M. Harman, and X. Yao. Software Module Clustering as a Multi-Objective Search Problem. *IEEE Transactions on Software Engineering*, pages 264–282, 2011.

[31] C. Prasad and W. Schulte. Taking Control of Your Engineering Tools. *Computer*, pages 63–66, 2013.

[32] C. Rossi. Release Engineering at Facebook. QCon San Francisco, http://goo.gl/b5LY80, 2012.

[33] G. Rothermel and M. J. Harrold. Analyzing Regression Test Selection Techniques. *IEEE Transactions on Software Engineering*, 22(8):529–551, 1996.

[34] G. Rothermel and M. J. Harrold. Empirical Studies of a Safe Regression Test Selection Technique. *IEEE Transactions on Software Engineering*, 24:401–419, 1998.

[35] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen. Build Code Analysis with Symbolic Evaluation. In *Proceedings of the 34th International Conference on Software Engineering (ICSE)*, pages 650–660, 2012.

[36] A. Telea and L. Voinea. A Tool for Optimizing the Build Performance of Large Software Code Bases. In *Proceedings of the 12th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 323–325, 2008.

[37] J. Thomas and A. Kumar. Google Engineering Tools. [Blog post] http://goo.gl/zOpl1T, 2011.

[38] V. Tzerpos and R. C. Holt. ACCD: An Algorithm for Comprehension-Driven Clustering. In *Proceedings of the 7th Working Conference on Reverse Engineering (WCRE)*, pages 258–267, 2000.

[39] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni. Automated Decomposition of Build Targets (Extended Version). http://hdl.handle.net/2142/47551, 2014.

[40] Apache Ant. http://ant.apache.org/.

[41] Apache Maven. http://maven.apache.org/.

[42] GNU Make. http://www.gnu.org/software/make/.

[43] Gradle. http://www.gradle.org/.

[44] Hudson. http://hudson-ci.org/.

[45] Jenkins. http://jenkins-ci.org/.

[46] Maven Central Repository. http://search.maven.org/.

[47] Google Protocol Buffers: Google's Data Interchange Format. Documentation and open-source release https://developers.google.com/protocol-buffers/.

[48] Rake. http://rake.rubyforge.org/.

[49] Travis. https://travis-ci.org/.

[50] T. Wiggerts. Using Clustering Algorithms in Legacy Systems Remodularization. In *Proceedings of the 4th Working Conference on Reverse Engineering (WCRE)*, pages 33–43, 1997.

[51] J. Wu, A. E. Hassan, and R. C. Holt. Comparison of Clustering Algorithms in the Context of Software Evolution. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, pages 525–535, 2005.

[52] N. York. Build in the Cloud: Accessing Source Code, 2011. [Blog post] http://goo.gl/H9WUGe.

[53] J. Zheng, B. Robinson, L. Williams, and K. Smiley. Applying Regression Test Selection for COTS-based Applications. In *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, pages 512–522, 2006.