Forecasting the Duration of Incremental Build Jobs

Qi Cao*, Ruiyin Wen†, Shane McIntosh†
*Yale University, USA; qi.cao@yale.edu
†Software REBELs, McGill University, Canada; ruiyin.wen@mail.mcgill.ca, shane.mcintosh@mcgill.ca

***Abstract*—Build systems automate the process of compiling, testing, packaging, and deploying modern software systems. While building a simple program may only take a few seconds on most modern computers, it may take hours, if not days, to build large software systems. Since modern build tools do not provide estimates of how long a build job will take, development and release teams cannot plan human and computer resources optimally. To fill this gap, we propose BUILDMÉTÉO—a tool to forecast the duration of incremental build jobs. BUILDMÉTÉO analyzes a timing-annotated Build Dependency Graph (BDG) that we extract from the build system to forecast build job duration. We evaluate BUILDMÉTÉO by comparing forecasts to the timed execution of 2,163 incremental build jobs derived from replayed commits of the GLIB and VTK open source systems. We find that: (a) 87% of the studied commits do not change the BDG, suggesting that reasoning about build job duration using the BDG is a sensible starting point; (b) 94% of incremental build jobs that do not change the BDG have an estimation error of under ten seconds; and (c) build jobs with larger sets of modified files tend to yield more accurate duration forecasts. These results suggest that BUILDMÉTÉO can improve the transparency of build jobs, and thus, aid practitioners in build-related decision making.**

I. INTRODUCTION

Build systems specify how project artifacts like source code, libraries, and data files are transformed into deliverables like executables. To achieve this transformation, build systems orchestrate hundreds of order-dependent commands that invoke compilers, interpreters, and testing tools to name a few.

Build systems play a pivotal role in modern software organizations. Developers rely on the build system to produce testable versions of deliverables after modifying the source code. Continuous integration and continuous delivery techniques rely on the build system to provide a rapid feedback loop for development teams and system users, respectively [6].

For large systems, executing a build job can take hours or even days to complete. For instance, build jobs of the Chromium web browser take more than one hour to complete.[1] Hassan and Zhang [5] find that certification build jobs of a large IBM system take more than 24 hours to complete.

To avoid re-executing every build-invoked command, the cornerstone feature of any build system is the *incremental build*, where the minimal set of commands that are required to propagate the code changes to deliverables are re-executed. Naturally, some incremental build jobs invoke more commands than others. For example, since a change to a header file will trigger recompilation of all of the files that include it, a change to a frequently included header file will trigger a long incremental build job. Conversely, a change to an infrequently included file will trigger a short incremental build job.

Unfortunately, build tools do not forecast the duration of incremental build jobs. The lack of forecasts has two negative implications. First, it creates a lack of transparency—developers do not know how long their build jobs will take to complete. While waiting for build jobs to complete, developers either: (a) switch contexts, which may be frustrating and wasteful if the switch is not necessary [9]; or (b) are effectively idle. Second, load balancing within fleets of build infrastructure must be load agnostic. As such, some nodes in the fleet may become overworked, while others are underutilized. Reactionary rebalancing of the build load can be used, but this requires a problem to occur before an action is taken.

This paper presents BUILDMÉTÉO[2]—to the best of our knowledge, the first tool to forecast the duration of an incremental build job at initiation-time. These forecasts will help developers and release engineers to make more informed decisions about when to switch contexts and how to balance the build load. To produce forecasts, BUILDMÉTÉO extracts the Build Dependency Graph (BDG) of the system being built and annotates nodes in the graph with timing information. When provided with a set of modified files, BUILDMÉTÉO selects the impacted nodes from the BDG and aggregates them to reason about the duration of that incremental build job.

To evaluate BUILDMÉTÉO, we compare the forecasted and actual duration of 2,163 incremental build jobs derived from historical commits of the GLIB and VTK open source systems. Preliminary analysis shows that 87% of the studied commits do not change the BDG, suggesting that our approach to reason about build job duration using the prior version of the BDG is a sensible starting point. Moreover, using the replay data, we address the following research questions:

RQ1. How accurately can BUILDMÉTÉO forecast the duration of an incremental build job?
The forecasts of 94% of the studied non-build-changing commits are within ten seconds of the actual duration.

RQ2. Is there a relationship between BUILDMÉTÉO's accuracy and the size of a code change?
As the number of modified files in a commit increases, estimation error tends to decrease (Spearman's $\rho = 0.63$, $p < 2.2 \times 10^{-16}$).

To aid in future replication studies, the source code of BUILDMÉTÉO and the experimental data is available online.[2]

[1]https://groups.google.com/a/chromium.org/forum/#!msg/chromium-dev/ebnkCESeTK4/tJ_ZS_nhAgAJ

[2]https://github.com/software-rebels/buildmeteo

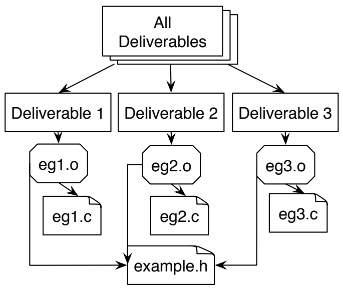
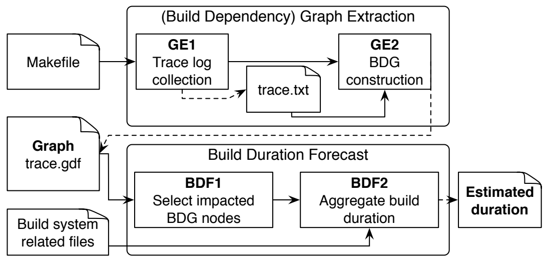


Fig. 1. An example Build Dependency Graph (BDG).



(Build Dependency) Graph Extraction
Makefile
GE1 Trace log collection
trace.txt
GE2 BDG construction
Graph trace.gdf
Build Duration Forecast
BDF1 Select impacted BDG nodes
BDF2 Aggregate build duration
Build system related files
Estimated duration

Fig. 2. The BUILDMÉTÉO approach to forecast build job duration.

The rest of the paper is organized as follows. Section II provides more detail about incremental building. Section III describes our approach to forecast the duration of incremental build jobs. Section IV presents the design of our commit replay study, while Section V presents the results. Section VI surveys related work. Section VII the discloses threats to the validity of our study. Finally, Section VIII draws conclusions.

II. INCREMENTAL BUILDING

With modern build tools, a full build job only needs be performed once, i.e., after the initial checkout of a codebase from the version control system. Subsequent build jobs can be *incremental*, only re-executing the commands that are required to propagate changes to the codebase since the prior build.

By only re-executing the required part of the full build process, incremental build jobs save plenty of time. Figure 1 presents an example of a BDG, which is used by the build tool to reason about which commands need to be re-executed. For example, after executing a full build job, if a developer changes `eg1.c`, an incremental build job only needs to recompile `eg1.o` and re-link `Deliverable 1`.

In a small system, incremental build jobs will only omit a few commands; however, as systems grow and the BDG expands, the time that is saved by incremental build jobs will become considerable. Yet changes to heavily reused code will trigger long incremental build jobs. For example, if `example.h` is modified, the incremental build will recompile `eg1.o`, `eg2.o`, and `eg3.o`, and also re-link `Deliverable 1`, `Deliverable 2`, and `Deliverable 3`. In this case, the incremental build job is identical to a full build job.

Since build tools do not forecast the duration of an incremental build job, practitioners do not know whether a job will take seconds, minutes, or even hours. This paper presents BUILDMÉTÉO, which can provide these forecasts.

III. BUILDMÉTÉO

In this section, we describe how BUILDMÉTÉO forecasts the duration of incremental build jobs. Figure 2 provides an overview of the approach, which is comprised of BDG extraction and duration forecasting steps (described below).

A. Build Dependency Graph Extraction

We use MAKAO [1] to extract the BDG from a software system. MAKAO constructs the BDG by parsing trace logs from a full build job. Thus, to extract the BDG, we collect a trace log from a full build job and feed it to MAKAO.

(GE1) Trace log collection. To generate the necessary trace logs, we execute a full build job for a system under test with GNU `make` tracing enabled. Furthermore, we instrument the shell that is spawned by the `make` process in order to compute the time that elapses during the execution of each build command. The trace log contains: (1) a listing of the commands that were executed during the build job; (2) the time that elapsed during the execution of each invoked command in the build job; and (3) the data that was used to reason about whether each command should be invoked or omitted.

(GE2) Build dependency graph construction. The trace log that we generate in step (GE1) is then fed to the MAKAO tool [1]. MAKAO parses the trace log to construct the BDG. Note that the BDG that we construct in this paper is a variant of the basic MAKAO BDG [1], which now contains the timing information that is needed to forecast build job duration. More specifically, this timing-annotated BDG contains:

- Graph nodes that denote build targets T. Each node $t \in T$ includes measurements of the time that elapsed when the commands that are associated with target t were invoked. Since varying load on our machines may influence our measurements, we repeat the measurements ten times and select the median elapsed time for each target.
- Directed edges that denote dependencies $d(t', t) \in D$ from target t' to target t. A dependency exists between targets t' and t if the commands of t' must also be invoked whenever the commands of t are invoked.

B. Build Duration Forecasting

Next, we forecast the duration of build jobs by analyzing the BDG. The process is comprised of the following two steps.

(BDF1) Select impacted BDG nodes. Given a set S of modified files, for each file $s \in S$, we identify the set of nodes $n \subset T$ that will be triggered should s change. To do so, we follow all of the edges $d(s', s) \in D$ that directly or transitively depend upon s in the BDG and add it to a subgraph R. We repeat this process for each file $s \in S$ such that R contains the subgraph of triggered nodes and edges $\forall s \in S$.

(BDF2) Aggregate build job duration of the triggered nodes. In order to forecast build job duration, we traverse and analyze R. In our current implementation of BUILDMÉTÉO, we sum up the timing measurements of each node $r \in R$ to generate the duration forecasts.

IV. Replay Study Design

We now describe the studied systems and our approach to evaluate BuildMétéo using a commit replay study.

A. Studied Systems

In order to address our research questions, we preform a replay study using historical commits from the GLib and VTK systems. GLib is a core library used in several GNOME applications.[3] The Visualization ToolKit (VTK) is used to generate 3D computer graphics and process images.[4]

We select GLib and VTK because their build processes are large and complex, having the capacity to trigger incremental build jobs of varying duration. Moreover, the GLib and VTK systems are actively maintained by their communities, generating plenty of commits for replay analysis.

B. BDG Extraction

In order to extract the BDG, we need to prepare a build environment first. To prepare a build environment for GLib, we first clone the GLib repository.[5] Then, after installing the required packages (i.e., libiconv, gettext, libffi, libintl), we invoke the `autogen.sh` script to prepare the codebase for build execution. Similarly for VTK, we clone its repository[6] and configure the fresh checkout for build execution by invoking the `cmake` command.

C. Replaying Commits as Incremental Build Jobs

We forecast the duration of incremental build jobs for 809 commits between versions 2.45.1 and 2.49.1 of GLib and 1,354 commits between versions 6.3 and 7.0 of VTK. For each commit, the replay analysis yields forecasts of the duration of an incremental build job for that commit, as well as actual measurements of the duration of the incremental build job.

The replay study begins by checking out the first (oldest) commit in our local build environment. We then perform a traced full build job and construct the BDG (see Section III-A). Next, we extract the list of modified files S from the commit under test and feed them to our module for forecasting build job duration. This will yield an estimate of how long the incremental build job for this commit will take. Then, we apply the `touch` command to all files $s \in S$, which updates the last modified time of each of the files, and will force subsequent incremental build jobs to invoke the build commands that are associated with those files. Finally, we use GNU `time` to measure the actual time that elapses during that incremental build job. After collecting both the forecast and the actual measurement for the commit, we checkout the next commit in the list and repeat the process.

Parallelism setting. Modern build tools allow the user to specify a parallelism setting (e.g., the `-j<N>` option of `make`), which allows up to the specified number of commands to be simultaneously invoked. In practice, the structure of the BDG imposes limits on parallelism. For example, in Figure 1, the paths `eg1.o` $\leftarrow$ `Deliverable 1`, `eg2.o` $\leftarrow$ `Deliverable 2`, and `eg3.o` $\leftarrow$ `Deliverable 3` can be executed in parallel. Regardless of the parallelism setting, at most three commands can be invoked in parallel for this BDG.

In our replay study, we use a parallelism setting of one (i.e., serial execution)—the default build behaviour of the studied systems. In future work, we plan to extend BuildMétéo to other parallelism settings by identifying sets of `N` parallel paths (where `N` is the parallelism setting) and using the duration of the longest path as a forecast for the duration of those paths.

[3]https://developer.gnome.org/glib/
[4]http://vtk.org/
[5]https://github.com/GNOME/glib
[6]https://gitlab.kitware.com/vtk/vtk.git

V. Replay Study Results

We now present the results of our replay study with respect to a preliminary analysis and our two research questions.

A. Preliminary Analysis of BDG Modification

BuildMétéo relies on the prior version of the BDG to generate forecasts. If the build job includes changes to the BDG, the duration forecasts will be unreliable. Hence, we want to know how often the studied commits modify the BDG.

Approach. Similar to prior work [2, 8], we semi-automatically label each file from the studied commits as build-related or not. In the first pass, we use file naming conventions to identify the build-related files (e.g., `Makefile`, `CMakeLists.txt`). We then perform a second, manual pass over the non-build files and identify some additional non-traditional build files. This approach is not perfect, and may introduce noise in our analysis. We discuss this threat to validity in Section VII.

Observations. Observation 1—In total, 87% of the studied commits do not modify the build system. We find that 89% of the studied GLib commits do not change the build system (and thus, do not update the BDG). Moreover, 86% of the studied VTK commits do not change the build system.

Note that these values are an upper bound of BDG-modifying commits. Indeed, a change to a build file does not necessarily update the BDG (e.g., a cosmetic change). Moreover, a build system change may modify an unrelated region of the BDG, which would not impact the forecast.

Since only 13% of the studied commits are build-changing, using the prior version of the BDG to forecast build job duration is a sensible starting point.

(RQ1) How accurately can BuildMétéo *forecast the duration of an incremental build job?*

Approach. To address RQ1, we obtain the forecasted and actual build job duration of each of the studied commits. Recall that if the build specifications change, the BDG may need to be updated and our forecasts may be unreliable (see Section V-A). Therefore, we group the studied commits into build-changing and non-build-changing categories. Finally, we calculate the *estimation error*, i.e., the difference between the forecasted and actual duration of incremental build jobs.

Observations. Figure 3 shows the distribution of estimation error values of the studied commits.

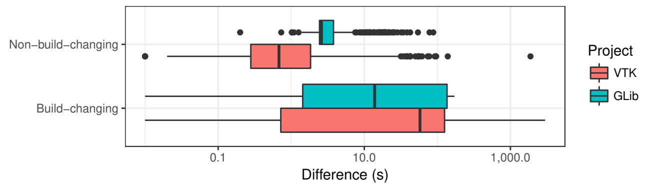


Fig. 3. Estimation error values for each of the studied commits (in log scale).

Observation 2—The estimation error of non-build-changing commits is small. Figure 3 shows that 93% and 95% of the non-build-changing commits have an estimation error of less than ten seconds in GLIB and VTK, respectively. Note that the average build time for non-build-changing commits in GLIB and VTK is 21 seconds and 47 seconds, respectively.

On the other hand, Figure 3 shows that 5% of commits in GLIB and 9% of commits in VTK have estimation error values greater than 50 seconds. Such large discrepancies exist because some header files are carefully refactored to reduce dependencies on them. For example, commit `1eedaf8` of the VTK project is a commit where the `vtkBitArray.h` header file was refactored to sever a dependency to `TestArrayLookup.cxx.o`. Since changes in `TestArrayLookup.cxx.o` also trigger 76 other targets to be invoked, this change has a large impact on the forecasted build job duration. More specifically, since BUILDMÉTÉO generates its forecasts based on the prior structure of BDG, it presumes that the severed dependency still exists, and forecasts a longer build job duration than we measured.

Observation 3—Even for build-changing commits, the estimation error is often low. In GLIB, 49% of build-changing commits have an estimation error of less than ten seconds. In VTK, 31% have an estimation error of less than ten seconds. Note that the average build time for build-changing commits in GLIB and VTK is 47 seconds and 377 seconds, respectively. Although BUILDMÉTÉO relies on the prior structure of the BDG, the duration of build-changing commits can still be accurately forecasted in many cases.

BUILDMÉTÉO*'s forecasts are within an error margin of ten seconds for 93%–95% of non-build-changing commits and 31%–49% for build-changing commits.*

(RQ2) Is there a relationship between BUILDMÉTÉO*'s accuracy and the size of a code change?*

Approach. To address RQ2, for each commit, we compare the estimation error with the size of the change in terms of the number of modified files. We then study the relationship between estimation error and the number of modified files.

Results. Figure 4 shows the estimation error plotted against the number of modified files in each studied commit.

Observation 4—For the non-build-changing commits, as the number of modified files increase, estimation error tends to decrease. Figure 4 shows the declining tendency of estimation error. Spearman's rank correlation tests monotonic relationships between variables regardless of their linearity.

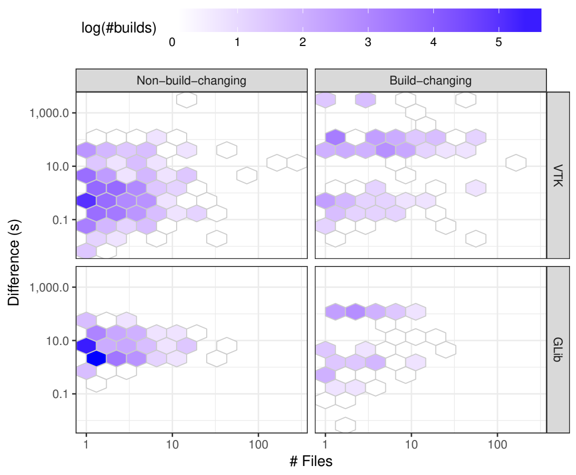


Fig. 4. A plot of estimation error against the number of modified files for each studied commit (in log scale).

Indeed, a Spearman correlation test between estimation error and the number of modified files yields a ρ value of 0.63 ($p < 2.2 \times 10^{-16}$). However, it does not necessarily mean that the forecasts for jobs that modify few files are imprecise. For example, we find that 95% and 94% of the commits that only change one file have an estimation error of less than ten seconds in GLIB and VTK, respectively.

Observation 5—For build-changing commits, there is no clear correlation between the difference and the number of modified files. Again, since the BDG may be in flux when build specifications are modified, the number of modified files does not have a clear impact on estimation error values.

For non-build-changing commits, the forecasting accuracy of BUILDMÉTÉO *tends to improve as the number of modified files increases. For the build-changing commits, the number of modified files is not related to estimation error values.*

VI. Related Work

Recent work has proposed several tools to support developers and release engineers in developing and maintaining build systems. Adams *et al.* [1] develop the MAKAO tool for visualizing, querying, filtering, refactoring, and validating build systems. Tamrawi *et al.* [10] introduce SYMake, an infrastructure for analysis of `make` specifications, and used it to detect code smells and support refactoring activities. Al-Kofahi *et al.* [3] present MkDiff to detect semantic changes in build files. In this paper, we also propose a tool; however, rather than supporting build system understanding and maintenance, BUILDMÉTÉO forecasts build job duration to improve the transparency of incremental build jobs.

Other work aims to predict the outcome of a build job. For example, Hassan and Zhang [5] use decision trees to predict whether a build job will pass a certification testing suite. Wolf *et al.* [12] and Kwan *et al.* [7] predict build job failures using *socio-technical congruence*, i.e., the agreement between social and technical project dependencies (or lack thereof). We do not aim to predict the outcome of a build job, but instead focus on forecasting the duration of incremental build jobs.

Speeding up build jobs has also been the focus of previous literature. Yu *et al.* reduce compilation time by analyzing syntactic dependencies in fine-grained program units to remove redundancies [14] and unnecessary dependencies [13]. Dayani-Fard *et al.* [4] introduce a targeted architectural restructuring technique that aims to speed up build jobs. Vakilian *et al.* [11] introduce a greedy algorithm that decomposes underutilized targets so that incremental build jobs do not trigger unnecessary build activity. While the prior work aims to improve the performance of individual build jobs, in future work, we plan to use our forecasts of the duration of build jobs to improve organization-level build performance by more intelligently provisioning build jobs within fleets of build infrastructure.

VII. Threats to Validity

Construct validity. Job duration forecasts for changes that modify the BDG are often inaccurate. Indeed, estimation error tends to be larger in build-changing commits than non-build-changing commits. On the other hand, Section V-A shows that only 13% of changes have to potential to change the BDG. Moreover, BuildMétéo prints a warning when a change to a known build specification is detected. Nonetheless, an incremental approach to update BDGs would likely reduce the estimation error of build-changing commits.

Internal validity. As described in Section III, we semi-automatically detect build specifications. Some files that were not automatically classified based on file name conventions were classified manually using the authors' intuition from prior experiences with build systems. To facilitate future studies, we include the classification data in our online appendix.

External validity. We focus our replay study on commits from the GLib and VTK open source projects, which may threaten the generalizability of our results. While we select systems from different sizes and domains, replication studies are needed to arrive at more general conclusions.

GLib and VTK use `make`-based build tools to implement their build systems, which may bias our results towards such build tools. On the other hand, our duration forecasts are computed using the BDG—a common construct among build tools. In future work, we plan to extend BuildMétéo to apply to newer build tools like `bazel`[7] and `buck`.[8]

VIII. Conclusion

Incremental build jobs vary in terms of duration. Unfortunately, modern build tools do not provide a forecast of how long a job will take. Without forecasts, developers may make costly, unnecessary context switches (e.g., when the build job completes within seconds or minutes) or may wait for a build job to complete when a context switch would have been more prudent (e.g., when the build job takes hours to complete).

In this paper, we present BuildMétéo—a tool that forecasts the duration of incremental build jobs by analyzing a timing-annotated BDG. In evaluating BuildMétéo using 2,163 commits from the GLib and VTK open source projects, we make the following observations:

- At least 87% of the studied commits do not change the BDG, suggesting that using the prior version of the BDG to forecast build job duration is a sensible starting point.
- The forecasts are within ten seconds of the actual job duration for 94% of the non-build-changing commits.
- As the number of modified files increases, estimation error decreases (Spearman's $\rho = 0.63$, $p < 2.2 \times 10^{-16}$).

In future work, we will expand BuildMétéo to support additional parallelism settings. Moreover, we will build upon BuildMétéo to explore whether duration forecasts can be used to achieve a more balanced organization-level build load.

[7]https://bazel.build/
[8]https://buckbuild.com/

Acknowledgments

This work was supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

References

[1] B. Adams, H. Tromp, K. de Schutter, and W. de Meuter, "Design recovery and maintenance of build systems," in *Int'l Conf. on Software Maintenance (ICSM)*, 2007, pp. 114–123.

[2] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter, "The Evolution of the Linux Build System," *Electronic Communications of the ECEASST*, vol. 8, 2008.

[3] J. M. Al-Kofahi, H. V. Nguyen, A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Detecting Semantic Changes in Makefile Build Code," in *Int'l Conf. on Software Maintenance (ICSM)*, 2012, pp. 150–159.

[4] H. Dayani-Fard, Y. Yu, J. Mylopoulos, and P. Andritsos, "Improving the Build Architecture of Legacy C/C++ Software Systems," in *Int'l Conf. on Fundamental Approaches to Software Engineering (FASE)*, 2005, pp. 96–110.

[5] A. E. Hassan and K. Zhang, "Using Decision Trees to Predict the Certification Result of a Build," in *Int'l Conf. on Automated Software Engineering (ASE)*, 2006, pp. 189–198.

[6] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.

[7] I. Kwan, A. Schroter, and D. Damian, "Does Socio-Technical Congruence have an Effect on Software Build Success? A Study of Coordination in a Software Project," *IEEE Transactions on Software Engineering (TSE)*, vol. 37, no. 3, pp. 307–324, 2011.

[8] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan, "An Empirical Study of Build Maintenance Effort," in *Int'l Conf. on Software Engineering (ICSE)*, 2011, pp. 141–150.

[9] S. C. Müller and T. Fritz, "Stuck and Frustrated or In Flow and Happy: Sensing Developers' Emotions and Progress," in *Int'l Conf. on Software Engineering (ICSE)*, 2015, pp. 688–699.

[10] A. Tamrawi, H. A. Nguyen, H. V. Nguyen, and T. N. Nguyen, "Build Code Analysis with Symbolic Evaluation," in *Int'l Conf. on Software Engineering (ICSE)*, 2012, pp. 650–660.

[11] M. Vakilian, R. Sauciuc, J. D. Morgenthaler, and V. Mirrokni, "Automated decomposition of build targets," in *Int'l Conf. on Software Engineering (ICSE)*, 2015, pp. 123–133.

[12] T. Wolf, A. Schroter, D. Damian, and T. Nguyen, "Predicting Build Failures Using Social Network Analysis on Developer Communication," in *Int'l Conf. on Software Engineering (ICSE)*, 2009, pp. 1–11.

[13] Y. Yu, H. Dayani-Fard, J. Mylopoulos, and P. Andritsos, "Reducing Build Time Through Precompilations for Evolving Large Software," in *Int'l Conf. on Software Maintenance (ICSM)*, 2005, pp. 59–68.

[14] Y. Yu, H. Dayani-Fard, and J. Mylopoulos, "Removing False Code Dependencies to Speedup Software Build Processes," in *Conf. of the Centre for Advanced Studies on Collaborative Research (CASCON)*, 2003, pp. 343–352.