Noname manuscript No.
(will be inserted by the editor)

An Empirical Study of Unspecified Dependencies in Make-Based Build Systems

Cor-Paul Bezemer · Shane McIntosh · Bram Adams · Daniel M. German · Ahmed E. Hassan

Author pre-print copy. The final publication is available at Springer via:
`https://dx.doi.org/10.1007/s10664-017-9510-8`

Abstract Software developers rely on a build system to compile their source code changes and produce deliverables for testing and deployment. Since the full build of large software systems can take hours, the incremental build is a cornerstone of modern build systems. Incremental builds should only recompile deliverables whose dependencies have been changed by a developer. However, in many organizations, such dependencies still are identified by build rules that are specified and maintained (mostly) manually, typically using technologies like `make`. Incomplete rules lead to unspecified dependencies that can prevent certain deliverables from being rebuilt, yielding incomplete results, which leave sources and deliverables out-of-sync.

In this paper, we present a case study on unspecified dependencies in the `make`-based build systems of the GLIB, OPENLDAP, LINUX and QT open source projects. To uncover unspecified dependencies in `make`-based build systems, we use an approach that combines a conceptual model of the dependencies specified in the build system with a concrete model of the files and processes that are actually exercised during the build. Our approach provides an overview of the dependencies that are used throughout the build system and reveals unspecified dependencies that are not yet expressed in the build system rules.

Cor-Paul Bezemer · Ahmed E. Hassan
Queen's University, Canada
E-mail: bezemer@cs.queensu.ca, ahmed@cs.queensu.ca

Shane McIntosh
McGill University, Canada
E-mail: shane.mcintosh@mcgill.ca

Bram Adams
Polytechnique Montréal, Canada
E-mail: bram.adams@polymtl.ca

Daniel M. German
University of Victoria, Canada
E-mail: dmg@uvic.ca

During our analysis, we find that unspecified dependencies are common. We identify 6 common causes in more than 1.2 million unspecified dependencies.

Keywords Build systems · unspecified dependencies

1 Introduction

Build systems (such as those specified using `make` [5]) describe how source code, libraries and data files are transformed into deliverables (*targets*), such as executables that are ready for deployment or testing. They consist of a set of build files (e.g., *makefiles*) containing targets and the rules that specify how to generate targets from their dependencies. Initially, the build system performs a full build to compile all targets. Afterwards, for example upon a local code change or an update to the latest file revisions from the version control system, an *incremental build* can be used to rebuild only the targets of which at least one dependency has changed. Such incremental builds substantially reduce the time required to wait for a build to finish, which is important since build systems range in size from hundreds to thousands lines of code [16]. Recent research has shown that the maintenance of build systems exhibits similar characteristics as maintenance of source code [14].

The essential ingredient for incremental compilation, i.e., dependency management, is a unique concept for build systems, which makes them difficult for development teams to maintain. When build maintenance is neglected, inconsistencies between the build system and source code may be introduced [15]. These inconsistencies typically manifest themselves as *unspecified dependencies*, i.e., dependencies that should be specified in the build system, but are not. These unspecified dependencies are subtle and difficult to detect.

Unspecified dependencies are problematic for two main reasons. First, incremental builds will omit build commands even though they are necessary. These incremental builds are incorrect, since changes to the sources will not be completely reflected in the deliverables. Second, parallel execution of build commands (a technique that is often used to speed up slow incremental build processes) will not preserve the correct order of these commands [18]. Indeed, since build tools like `make` decide which rules can be executed in parallel by checking the specified dependencies, unspecified dependencies will not be taken into account. Hence, rules that should have been executed sequentially may be executed simultaneously, causing race conditions. The importance of specifying all dependencies in a build system is emphasized by Michael Chastain, the maintainer of the Linux build system, who states: *"correctness trumps efficiency"* [17].

For example, recently such an unspecified dependency was discussed and patched by developers on the *fa.linux.kernel* newsgroup [29]. In the build system of the Linux kernel, the target *uImage* depends on the target *zImage*, but there was no dependency explicitly specified. As a result, when the rule for the *uImage* target was executed in parallel, *uImage* was invalid at random times, depending on the execution order chosen by the build system. Since unspecified dependencies do not necessarily generate build error messages, and the build seemingly randomly generates incorrect deliverables, such problems can only be found during testing, at which time the link

to incorrect build specifications is hard to make. This bug was fixed by adding an explicit dependency for *uImage* on *zImage*.

Several build technologies, e.g., fabricate [11] and memoize [13], have been developed that are able to automatically resolve unspecified dependencies. While these build technologies may be more advanced than `make` with respect to dependency management, McIntosh et al. [16] show that `make`-based build systems are still by far the most often used. Hence, the projects that rely on these `make`-based build systems remain susceptible to unspecified dependencies.

In this paper, we present a case study on the unspecified dependencies in four popular open source projects (GLIB, OPENLDAP, LINUX and QT) that use `make`-based build systems. To enable our study of unspecified dependencies, we first present our approach for uncovering unspecified dependencies in `make`-based build systems. The foundation of our approach is a consolidated view that combines the *conceptual dependency model*, which is specified in the build system itself, with the *concrete dependency model*, which is implied by the read and write behaviour of the processes that are executed during the build process.

The main contributions of this paper are:

- An empirical analysis of unspecified dependencies in the build systems of four popular open source projects.
- A discussion of the root causes of over 1.2 million unspecified dependencies that we uncovered during our analysis.
- A theoretical foundation using Tarski algebra for detecting unspecified dependencies.
- A prototype tool that is built upon this theoretical foundation.

The outline of this paper is as follows. Section 2 provides background information, presents a motivational example for our work, and situates this paper with respect to the build systems literature. Section 3 presents our methodology and its implementation. Sections 4 and 5 describe our case study and its results. Section 6 discloses the threats to the validity of our study. In Section 7, we discuss the broader implications of our results. Finally, Section 8 draws conclusions.

2 Background

2.1 Build Systems

Build systems use rules to specify how *targets* must be (re)built. These rules define the dependencies that a target depends on (e.g., which files must exist before the target can be created), as well as the commands that are required to generate the target once its dependencies exist. These dependencies are used to keep a target up-to-date with the rest of the system — if a dependency changes, targets that depend on it must be rebuilt.

In this paper, we focus on `make`-based build systems because `make` is the most popular file-based build technology [16]. Our focus includes technologies that use

make as its backend, such as KitWare's CMake, GNU's Autotools or the LINUX kernel's build system (kbuild). These 'higher-level' build technologies generate makefiles from higher-level configuration files, which are easier to maintain for developers. Figure 1 demonstrates the process of using such a higher-level build technology to generate a makefile. The configuration is triggered by a script that takes a configuration template as input (e.g., the configure.ac file for autoconf). The configuration template specifies which variables need to be configured, such as the compiler or the libraries on which the build depends. During the configuration, the script automatically scans the system for the variables in the configuration template. After scanning the system, the variable values are entered in the Makefile-template (e.g., the makefile.in for automake) and a makefile is generated. The generated makefiles are used by make to build the system.

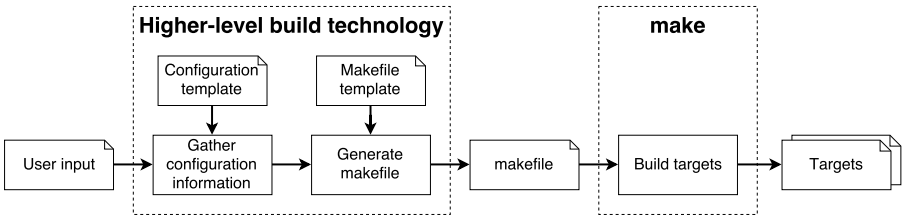


Fig. 1: The process of using a higher-level build technology to generate a makefile from a configuration file

In a file-based build system, such as make, targets are either files in the filesystem or are phony. A phony target does not produce a file directly, but instead is used to trigger other targets. The all target in Listing 1 is phony as it only triggers the app.o rule.

The build process is initiated by the make 'target' command. Then, for every dependency that causes a rule to be triggered, a child process is spawned to execute the commands that are specified in that rule. This results in a hierarchical tree of build processes.

Initially, a full build is performed to compile all of the targets that are specified in the build system. After that, incremental builds are usually performed, rebuilding only the targets of which at least one of the dependencies have changed. In large systems with long full build processes, incremental builds save developers' time by minimizing the idle time spent waiting for builds to complete.

Because all targets are rebuilt during a full build, they are usually correct and consistent. In an incremental build, problems may arise when targets have a rule with unspecified dependencies. In that case, a target is missing a dependency. If the missing dependency is modified, the target will not be rebuilt, even though it should be. In the next section, we will present a motivating example of a situation in which such *unspecified dependencies* become a problem.

2.2 Motivating Example

Listing 1 shows a makefile depicting targets `all`, `app.o` and `app.c` and their corresponding rules. The phony target `all` triggers the build for target `app.o`. The rule for `app.o` explicitly defines that `app.o` depends on `app.c` and `app.h` — only if `app.c` or `app.h` are updated, `app.o` must be rebuilt. The rule for `app.c` shows that it has no dependencies specified and that its contents are generated by `generator`.

Looking at the source code for `app.h` in Listing 2, we see that `app.h` includes `header.h`. Hence, if `header.h` changes, all targets depending on `app.h` should be updated as well. Listing 1 however, does not show any dependency on `header.h`. Therefore, `app.o` is not rebuilt when `header.h` is changed, causing `app.o` to remain outdated. The *unspecified dependency* on `header.h` from `app.o` is a bug in the build system.

In addition, `app.c` is generated by the `generator` script but it does not depend on `generator`. If the source code of `generator` changes, `app.c` is not regenerated, even though it should be. In this case, there is an unspecified dependency on `generator` from `app.c`, which is another bug in this build system.

```
1 all: app.o
2
3 app.o: app.c app.h
4     gcc -c app.c -o app.o
5
6 app.c:
7     ./generator -o app.c
```

Listing 1: Makefile

```
1 ...
2 #include "header.h"
3 ...
```

Listing 2: app.h

The current ‘industry standard’ for dealing with problems associated with unspecified dependencies is to remove all generated files and perform a full build (instead of an incremental one), or to run the incremental build multiple times until a fixed point is reached (no more rules are triggered) [18]. As described above, if `header.h` changes, the unspecified dependencies would cause `app.o` and `app.c` not to be rebuilt during an incremental build; but the full build recompiles all targets, regardless of what their dependencies are. While this approach may be feasible for smaller systems, it is not practical for large systems that take hours or even days [10] to perform a full build. In addition, because a full build is dependent on the build order of components, unspecified dependencies limit the ability of a build to be parallelized [18] or might lead to non-deterministic results.

In larger systems, this type of bug is difficult to detect. One of the reasons is that this type of bug cannot always be detected by inspecting the build file only, as demonstrated by Listings 1 and 2. In addition, in many modern systems, at least part of the build code is generated based on a template. As a result, if this template contains a bug, this bug can be found in many instances throughout the application, making it difficult to find all instances that share the same root cause. Finally, build systems change often [14, 15], which makes the risk of forgetting to add a dependency even larger.

The bugs in Listings 1 and 2 can be easily fixed by adding a dependency on `header.h` for `app.o` and on `generator` for `app.c`. The main problem is detecting such bugs in a larger build system, which requires an automated approach. In the remainder of this paper, we first present the approach that we will use to uncover unspecified dependencies in our empirical study of unspecified dependencies in `make`-based build systems.

2.3 Tarski Algebra

To avoid ambiguity in the presentation of our approach and of the findings of our empirical analysis, we formalize the notions of build dependency graphs and unspecified dependencies using Tarski's algebra [26]. Tarski's algebra defines a set of operators and algebraic rules for binary relations. The algebra's core concept is that of a relation R defined as $\{(x:X, y:Y)\} \in \mathscr{R}$, which corresponds to a set of tuples where entities $x:X$ are in a relation R with an entity $y:Y$. For example, we can define the relation $depends \subseteq file \times file$, where every tuple $(a,b) \in depends$ implies that a depends on b. The notation $n:t$ refers to an entity with identifier $n \in \mathscr{N}$ and a type $t \in \mathscr{T}$. Typically, the name of a relation, such as *depends*, is used as shorthand for the set of all tuples involved in that relation.

Relational algebra allows us to perform set operations on relations R_1 and R_2, such as taking set difference ($-$), union ($\cup$) and intersection ($\cap$):

$$\begin{aligned} R_1 - R_2 &= \{(x:X, y:Y) | (x:X, y:Y) \in R_1 \\ &\quad \wedge (x:X, y:Y) \notin R_2\} \\ R_1 \cup R_2 &= \{(x:X, y:Y) | (x:X, y:Y) \in R_1 \\ &\quad \vee (x:X, y:Y) \in R_2\} \\ R_1 \cap R_2 &= \{(x:X, y:Y) | (x:X, y:Y) \in R_1 \\ &\quad \wedge (x:X, y:Y) \in R_2\} \end{aligned}$$

Given the definition of a relation as a set of tuples, and the ability to combine relations using the union operator, one can then define a graph $G = \bigcup_{i \in \mathscr{S}_G} R_i$ with $R_i \in \mathscr{R}$. We call $\mathscr{S}_G$ the schema of graph G, as it specifies which relations (kinds of edges) are allowed. Basically, each edge of the graph corresponds to an entity tuple of a particular relation.

The transitive closure $R^+ \subseteq X \times X$ of a relation contains all the tuples $(x:X, y:X)$ that are on a directed path of $R \subseteq X \times X$ relations, i.e.:

$$R^+ = \{(x:X, y:X) | \exists n \in \mathcal{N} \geq 0 \; s.t. \\ (x:X, x_0:X), (x_0:X, x_1:X), ..., (x_n:X, y:X) \in R\}$$

2.4 Modeling Dependencies

Using Tarski's relational algebra, we model the dependency graph as a set of relations on two different sets: targets (which can be phony or files, which may or may not be created by the build process) and operating system processes (which read or write targets, e.g., files). We model the conceptual dependencies (i.e., those specified explicitly in the build system) as relations between two targets, while we model the concrete dependencies (i.e., those implied by read-write behaviour of processes that are executed by the build system) as relations between a process and a file. The intuition behind these two models is that the conceptual model contains the high-level dependencies between files and targets, while the concrete model covers every single file access during build execution, including all required files by a build. More specifically: $(a,b) \in \textit{depends}$ if a depends on b, as specified in a makefile; $(p,a) \in \textit{reads}$ if process p reads file a; $(a,p) \in \textit{writtenBy}$ if file a is written by process p; and $(p,q) \in \textit{executes}$ if process p executes process q.

To illustrate these concepts of Tarski's relational algebra, we use Figure 2 as an example. Figure 2 represents a conceptual model consisting of two rules: target x depends on phony target y and target y depends on file a. This conceptual model is exercised by process p that reads files a and b and writes x, and process q that executes process p (this part of the graph reflects the concrete model). Hence:

$$\begin{aligned} \textit{depends} &= \{(x,y),(y,a)\} \\ \textit{depends}^+ &= \{(x,y),(y,a),(x,a)\} \\ \textit{reads} &= \{(p,a),(p,b)\} \\ \textit{writtenBy} &= \{(x,p)\} \\ \textit{executes} &= \{(q,p)\} \end{aligned}$$

As described in Section 2.3, we can convert these relations into a graph $G = (V,E)$ as follows: assume F_s is the set of all the source files, F_t is the set of temporary files created and deleted during the build process, F_f the set of files that are explicitly specified and created during and kept after the build, and F_h the set of files that are not specified but created during the build and kept after the build; T will be the set of phony targets present in the makefiles of the system; finally, P is the set of processes executed during the build system. Hence:

$$\begin{aligned} V &= F_s \cup F_t \cup F_f \cup F_h \cup T \cup P \\ E &= \textit{depends} \cup \textit{reads} \cup \textit{writtenBy} \cup \textit{executes} \end{aligned}$$

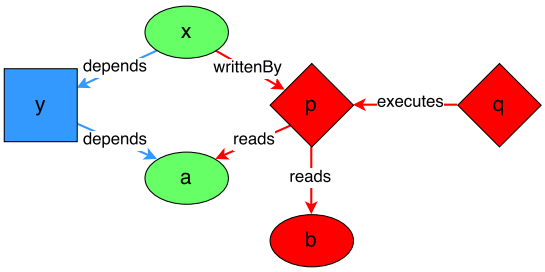


Fig. 2: Tarski algebra example (squares represent phony targets, ellipses represent files, diamonds represent processes, blue elements occur in the conceptual model only, red elements occur in the concrete model only and green elements occur in both models)

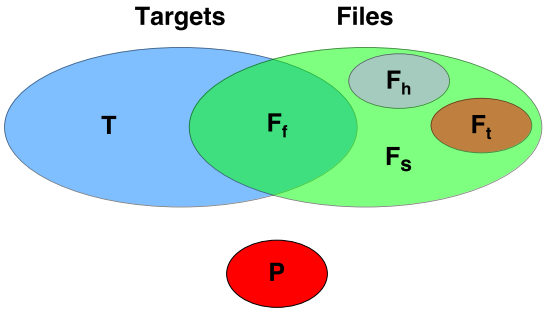


Fig. 3: Venn diagram of the relations between the sets used to create G

Each edge is labelled according to the relation set it belongs to. The resulting graph of the example above is depicted in Figure 2. Figure 3 shows how the sets of files and targets used to create G relate to each other.

We can use Tarski operations to identify properties of interest in this graph. For example, we can use set difference to determine the indirect transitive dependencies:

$$depends_{\text{indirect}} \triangleq depends^{+} - depends$$

which for our running example is $\{(x,a)\}$.

The compositional relation ($\circ$) is used to merge relations with each other:

$$R_1 \circ R_2 = \{(x:X, y:Y) | \exists z \in Z \; s.t. \; (x:X, z:Z) \in R_1 \\ \wedge (z:Z, y:Y) \in R_2\}$$

Using the compositional relation, we can find the pairs of all files that are being read and written by the same process (effectively extracting a $depends_{\text{actual}}$ relation from the concrete graph) as follows:

$$depends_{\text{actual}} = writtenBy \circ reads = \{(x,a),(x,b)\}$$

This relation matches each target with every file that is being read in order to produce the target.

2.5 Related Work

In this section, we describe the research that is related to the work presented in this paper, an analysis of unspecified dependencies in build systems.

Exploratory Studies Neitsch et al. [21] study the issues that arise in build systems of multilanguage software, i.e., software written in multiple programming languages. Neitsch et al. find that many such build systems require manual intervention to build the default targets, and also they present a set of (anti-)patterns that describe the observed issues.

Seo et al. [24] analyze 26.6 million builds to explore the causes, types of errors made and resolution efforts to fix the builds. Their findings show that approximately one-third of all executed builds fail and that most of these failures are caused by dependency issues.

Miller [18] present a type of bug that is caused by designing `make`-based build systems using recursion. `make` is often used this way, i.e., with a separate makefile in each directory, to keep makefiles manageable and lower build times for components. However, as Miller explains, separating makefiles forces recursive makefiles to become dependent on the execution order of each other. As discussed in Section 2.2, recursive makefiles limit the parallelizability of the build when there are unspecified dependencies in the build system.

Unspecified Dependency Detection in Build Systems Gunter [9] presents a method for simulating a model of a makefile using a petri-net. The petri-net model is used to check the correctness of and detect possible optimizations in a makefile. Gunter's model of a makefile is static, as it uses makefile-level information only. Therefore, it is not possible to detect all unspecified dependencies using Gunter's model (or any other static models), as detecting unspecified dependencies requires both makefile-level information (i.e., to create the conceptual graph) and execution-level information (i.e., to create the concrete graph), which our Tarski model captures. A clear example of data that is not captured by Gunter's model are temporary files that are generated by commands (as there are no explicit targets defined for such files).

Jørgensen [12] extracts a semantic model of a makefile using formal methods. The formal model is used to check the safeness of a `make` build system. In Jørgensen's model, the makefile is checked for rule completeness, fairness and soundness. Tamrawi et al. [25] present `SYMake`, a tool that creates a symbolic dependency graph from a `make` build system. `SYMake` can be used to detect build code smells such as cyclic dependencies and duplicate prerequisites.

Zhou et al. [28, 30] statically create a dependency graph of the makefile and the source code and compare these to find unspecified dependencies. A disadvantage of their approach is that static analysis on source code may miss dependencies that are dynamically loaded. The approach used in this paper is capable of finding unspecified dependencies that are dynamically loaded.

Nadi and Holt [19, 20] present an approach for detecting anomalies in the variability configuration in the LINUX kernel. Nadi and Holt check for cross and self-consistency between the compilation, configuration and implementation spaces.

These approaches all focus on finding, among other types of issues, unspecified dependencies in build systems. However, these approaches focus solely on the conceptual dependency model to search for unspecified dependencies. As a result, these static approaches do not uncover all unspecified dependencies in the build system that do not exist in this conceptual dependency model. One advantage that the static approaches have compared with the approach used in this paper, is that the static approaches are capable of analyzing all configurations of the build system in one run, while our approach requires a dynamic execution of every configuration.

`apmake` [4] is a tool that manages the order of executions in a build to make it parallelizable. Using timestamps and transaction-like behaviour, `apmake` provides concurrency control for `make`-based build systems to make sure that targets do not read from outdated targets as they are building. While `apmake` focuses on execution order to address the problem of unspecified dependencies, we focus on finding the actual unspecified dependencies. However, the approach used by `apmake` to decide upon the safest execution order is similar (i.e., it uses `ptrace`) to our approach for extracting the concrete dependency model.

Several modern build technologies, such as fabricate [11] and memoize [13] avoid the issue of unspecified dependencies altogether. Build systems that use fabricate or memoize do not specify dependencies. Instead, the dependency graph of the build system is extracted automatically at build time using the concrete dependency model.

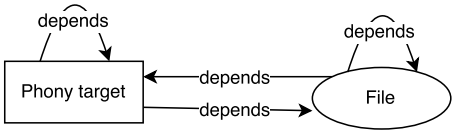
Google's Bazel build system [7] detects unspecified dependencies as well. However, Bazel's documentation states that: "The build tool attempts aggressively to check for missing dependencies and report errors, but it is not possible for this checking to be complete in all cases" [8]. The Bazel documentation does not state how Bazel checks for missing dependencies or why this check may be incomplete.

The fabricate, memoize and Bazel build technologies require the build system to be developed in that technology. The approach that we use during our empirical analysis is similar to the techniques of fabricate and memoize (i.e., using `strace`), but our approach works for existing `make`-based build systems. Hence, our approach is capable of finding unspecified dependencies in `make`-based build systems, which are widespread and the most mature and popular type of build system [16].

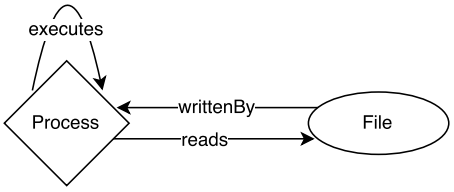
We are the first, to the best of our knowledge, to conduct an empirical analysis of unspecified dependencies in `make`-based build systems of large open source projects. In the next section, we first present our approach for automatically uncovering unspecified dependencies in `make`-based build systems.

3 Discovering Unspecified Dependencies

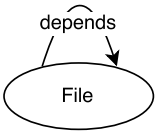
Our approach for detecting unspecified dependencies is based on identifying the differences between two graphs: 1) the *conceptual* dependency graph with the targets, files and dependencies that are explicitly mentioned in the build files and 2) the *concrete* dependency graph with all processes and files that are being executed during the build. However, since both graphs have a different schema, we first need to transform their schema into a common schema that we call the "consolidated" dependency schema. Without transforming the schema of the graphs into the consolidated depen-



(a) Conceptual schema



(b) Concrete schema



(c) Consolidated schema

Fig. 4: Schemas for dependency graphs

dency schema, we cannot compare what is supposed to happen (i.e., the conceptual graph) with what actually happened (i.e., the concrete graph) and hence, we cannot detect unspecified dependencies.

3.1 Schemas

Figures 4a and 4b show the possible schemas of the conceptual and concrete dependency graphs. Conceptual graphs contain the dependency information that is specified in a makefile, i.e., dependencies between build targets. Since a target either corresponds to a file like `app.o` or a phony target, representing a virtual build activity such as `compile` or `package`, the schema of a conceptual dependency graph only allows tuples of the $depends \subseteq target \times target$ relation, with $target = phony \cup file$. We define the conceptual graph $G_{conceptual}$ as follows:

$$
\begin{aligned}
G_{conceptual} &= (V_{conceptual}, E_{conceptual}) \\
V_{conceptual} &= F_s \cup F_f \cup T \\
E_{conceptual} &= depends
\end{aligned}
$$

with F_s, F_f and T as defined in Section 2.4.

On the other hand, the concrete dependency graphs model what actually happens during the execution of a build, i.e., the sequence of nested processes (tool invocations) and their file reading and writing behaviour. Nodes can only be of types

file (e.g., `app.o`) or *process* (e.g., `gcc`), since the concept of phony targets does not exist at run-time (only in the makefiles). The graph's schema permits the relations $reads \subseteq process \times file$, corresponding to files that are read during the build, $writtenBy \subseteq file \times process$, corresponding to files that are written and $executes \subseteq process \times process$, corresponding to processes that are executed. The concrete graph $G_{concrete}$ can be defined as follows:

$$G_{concrete} = (V_{concrete}, E_{concrete})$$
$$V_{concrete} = F_s \cup F_t \cup F_f \cup F_h \cup P$$
$$E_{concrete} = reads \cup writtenBy \cup executes$$

with F_s, F_t, F_f, F_h and P as defined in Section 2.4.

The resulting conceptual and concrete dependency graphs for the example of Listing 1 are shown in Figure 6. As we can see, the conceptual graphs and the concrete graphs contain different entities and relations. In the remainder of this section, we will explain how we can abstract both graphs to a consolidated schema, which we use to detect unspecified dependencies.

3.2 Abstracting Dependency Graphs Based on the Consolidated Schema

The conceptual and concrete dependency graphs in Figure 6 cannot be compared to each other, because they have different schemas, and hence, contain different types of relations and entities. In order to make them comparable, we define the consolidated schema as shown in Figure 4c. All entities are of type *file*, and the only allowed relation is $depends \subseteq file \times file$, which captures all required dependencies between any file that is used and/or generated by the build process of a project. After transforming the conceptual and concrete dependency graphs to this consolidated schema, resulting in the conceptual^A and concrete^A graphs, we can then compare concrete^A to conceptual^A to identify those nodes and edges that are used during build execution, but were not specified in the makefiles.

Obtaining conceptual^A requires removing *phony* targets and any dependencies on and of those targets. We remove *phony targets* using the following iterative algorithm:

1. For a file f, a phony target t, and a file or phony target c, if $(f,t) \in depends$ and $(t,c) \in depends$ then replace them with a new relation (f,c) such that $(f,c) \in depends$
2. Repeat step 1 until there are no more changes

The resulting graph only has *file* nodes.

In order to obtain concrete^A, we need to derive the *depends* relation from the *reads* and *writtenBy* relations. Indeed, each rule in a build file corresponds to the rule's target (*file*) that is generated by (*writtenBy*) a *process*. Hence, whenever we have a *process* node that writes a particular *file*, we know that there should be a *depends* relation between that *file* and all of the *file* nodes the *process* or its nested

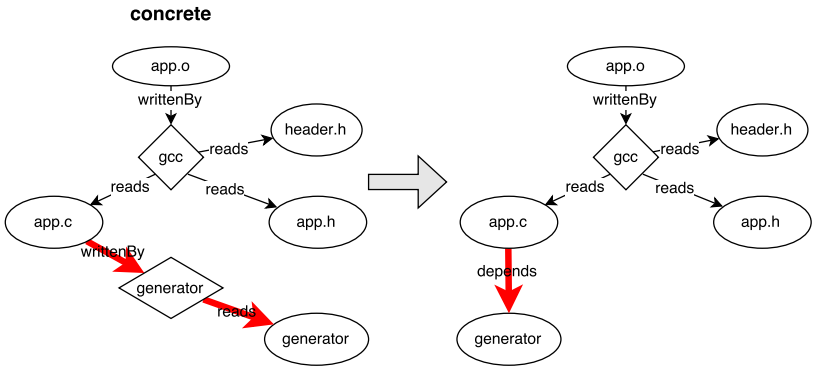


Fig. 5: Abstracting a *reads* and *writtenBy* relation

*process*es *reads*. We can find this *depends* relation as the pairs of all files that are being read and written by the same process as follows:

$$depends^{\mathrm{A}} \triangleq writtenBy \circ reads$$

Figure 5 demonstrates how the *reads* and *writtenBy* relations for `app.c` and `generator` are abstracted into the *depends* relation. The resulting conceptual$^{\mathrm{A}}$ and concrete$^{\mathrm{A}}$ graphs of the example of Section 2.2 are depicted by Figure 6.

Given that conceptual$^{\mathrm{A}}$ and concrete$^{\mathrm{A}}$ have the same types of entities and relations, we can easily identify the unspecified dependencies graph $G_{\text{unspecified}}$ via the relation:

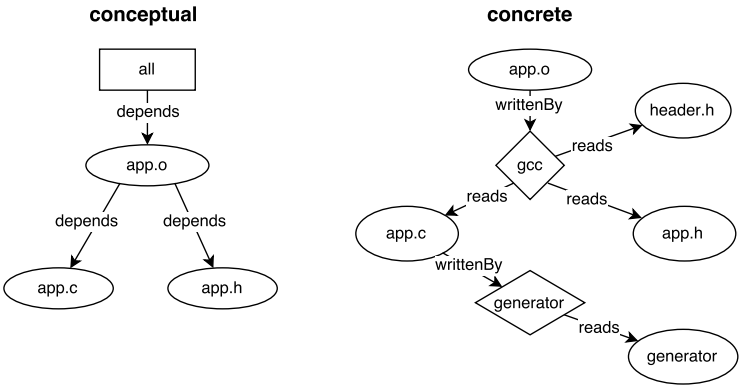
$$depends_{G_{\text{unspecified}}} \triangleq depends^{+}_{concrete^{\mathrm{A}}} - depends^{+}_{conceptual^{\mathrm{A}}}$$

In this example, $G_{\text{unspecified}}$ contains the dependencies (`app.o`, `header.h`) and (`app.c`, `generator`), which correspond to the unspecified dependencies that we identified in Section 2.2.

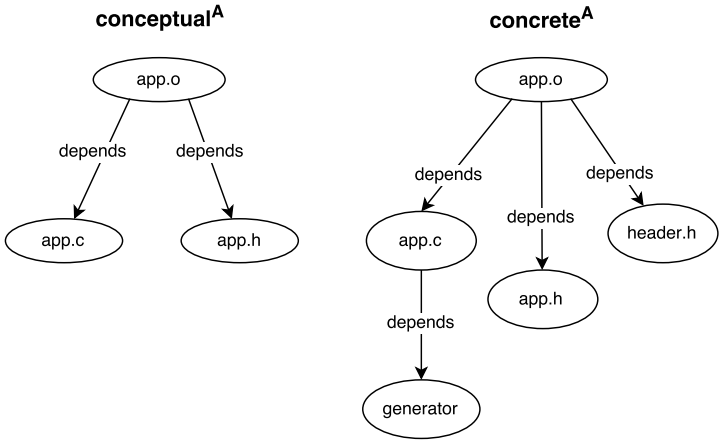
3.3 Implementation

Our approach is implemented in three phases: data extraction, graph abstraction and graph analysis. We have implemented our approach for `make`-based build systems, building on our prior research [1, 2] and the resulting toolset. Figure 7 depicts the steps taken.

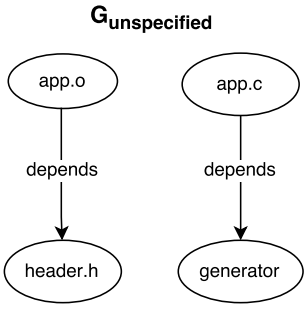
1. Data Extraction Our approach relies on the combination of two dependency graphs: 1) the conceptual dependency graph, which contains the targets and files that are explicitly defined in the build system and 2) the concrete dependency graph, which contains the set of processes and files that are exercised by the build process. As Figure 7 shows, we use MAKAO [1] to compute the conceptual dependency graph and we use BEE to compute the concrete dependency graph.



(a) Dependency graphs



(b) Abstracted dependency graphs



(c) Unspecified dependency graph

Fig. 6: Dependency graphs that correspond to the example in Section 2.2

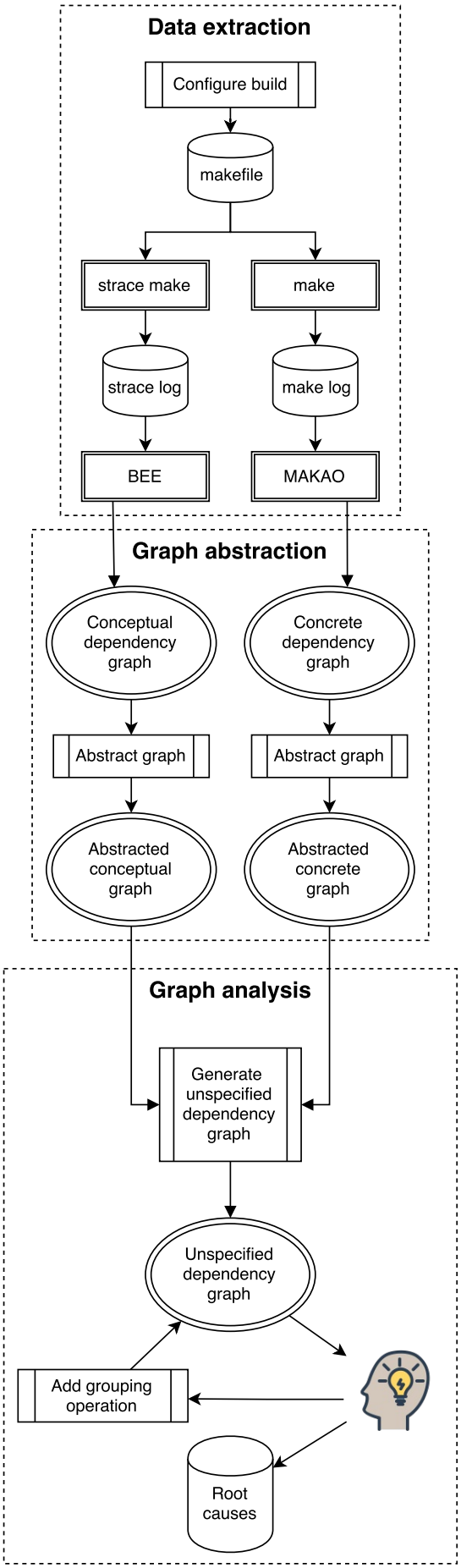


Fig. 7: Overview of our approach

To obtain the explicitly defined targets and files, we first configure our build for a particular set of configurable features. If the build system uses a higher-level build technology, such as `Autotools` or `kbuild`, we make sure that the makefile is generated from the configuration file before performing the build. We then perform a full build of the default target in the build system and record the verbose output of `make` into a log file.

We use MAKAO[1] to parse the output of the log file into a graph representation. This results in a graph database containing the explicitly defined target and file nodes and their dependencies. We use Neo4j[2] as our graph database. To facilitate a fast import of a large number of nodes and edges, we use the Neo4j Batch Importer.[3]

To obtain the concrete graph, we need to identify the set of processes and files that are used during the build. We run STRACE[4] during a full execution of the build to identify these processes and files. STRACE captures all system calls and events during the execution of a program into a log file. We parse the STRACE log using our Build Execution Explorer tool (BEE[5]). BEE parses the creation of new processes and maintains a list of currently executing processes. For every new process, a process node is appended to the graph. Then, for every read or write event, the graph is searched for the file being read or written. If a file node exists for the file, a read or write edge is created from the process node to that file. Otherwise, the file node is created together with the edge.

The parse time of our implementation is dominated by the parse time for the STRACE log, which is linear with the size of that log. As a result, the total parse time is linear with the execution time of the build. For projects that have a large parse time (e.g. several hours), our approach could run overnight to verify the nightly build without negatively affecting the development process.

2. Graph Abstraction To compare the conceptual and concrete graphs we must abstract them into conceptual^{A} and concrete^{A}. We perform the abstraction using queries on the Neo4j database.

3. Graph Analysis When detecting unspecified dependencies, we are mainly interested in their root cause, so that we can fix the root cause if necessary or recognize other unspecified dependencies of the same kind. By root cause we refer to the main reason for the unspecified dependency, which can be either unintentional (i.e., a bug in the build system) or intentional (e.g., a project policy). During the graph analysis phase of our methodology, we manually investigate root causes of unspecified dependencies. In many modern systems, at least part of the build system code is generated based on a template. As a result, if this template causes an unspecified dependency, many similar unspecified dependencies will occur throughout the build process.

[1] `http://mcis.polymtl.ca/makao.html`
[2] `http://neo4j.com/`
[3] `https://github.com/jexp/batch-import`
[4] `http://linux.die.net/man/1/strace`
[5] `https://github.com/smcintosh/bee`

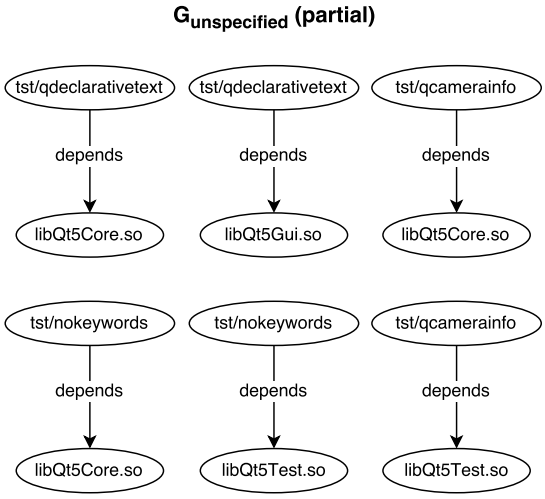


Fig. 8: Example of unspecified dependencies that can be grouped

In order to avoid having to manually analyze all these similar unspecified dependencies, we semi-automatically define grouping operations that cluster such dependencies in the unspecified dependency graph, allowing us to focus our analysis on unspecified dependencies for which a root cause has not yet been found. These grouping operations are defined during the analysis; when we encounter several similar unspecified dependencies, we define a grouping operation based on the commonalities in the filename of the dependencies or the process that loads the dependencies. We can then automatically query for all unspecified dependencies of a group.

Figure 8 shows a partial unspecified dependency graph. The commonality in the displayed unspecified dependencies is the extension of the unspecified dependencies (`*.so`). After identifying the root cause for one of these unspecified dependencies (see Section 5.3.2), there is no need to analyze other unspecified dependencies that clearly share the same root cause. Therefore, we define a grouping operation that groups all `*.so` dependencies from $G_{\text{unspecified}}$. Hence, while the grouping operations are not direct representations of the root cause, they are an invaluable tool to narrow down the number of unspecified dependencies that need to be analyzed manually during our study.

In the next section, we discuss the setup of an empirical case study of unspecified dependencies in the build systems of four open source projects.

4 Case Study Setup

We use our approach to conduct a case study on four open source projects. In this section, we describe the goal of our case study, our subject systems and our case study setup.

Case study goal In our case study, we focus on uncovering the following:

1. The unspecified dependencies in the studied open source projects
2. The root causes of the detected unspecified dependencies
3. The ways that developers deal with unspecified dependencies

In the remainder of this section, we discuss our subject systems and our case study setup.

Subject systems To conduct our empirical analysis of unspecified dependencies, we performed a case study on large and popular open source projects with a `make`-based build system. In selecting the subject systems, we identified two important criteria that needed to be satisfied:

- **Criterion 1 - Build technology:** The unspecified dependencies that we aim to detect are exclusive to build systems that are based on the `make` technology (note that this includes build systems that use `make` as backend, such as the `Autotools` and `kbuild` technologies that are mentioned in Section 2.1). Hence, we analyze projects that use the `make` technology only.
- **Criterion 2 - Impact:** We want to show that our approach can find unspecified dependencies in real applications that are widely used, as the impact of such findings can be the largest. In addition, the build system of these applications has been executed many times before, making it more likely that obvious issues with unspecified dependencies are fixed. Hence, we analyze projects with a large user-base only.

We analyze the subject systems that are described in Table 1. GLIB[6] is a bundle of low-level libraries, that provide advanced data structures. OPENLDAP[7] is an open source implementation of the Lightweight Directory Access Protocol. The LINUX kernel[8] forms the foundation of the operating system used by millions of users. QT[9] is a set of libraries and tools for building cross-platform applications.

The *build time* in Table 1 is the approximate time it takes to perform a complete build of the project. We include this measure to describe the size and complexity of the build system. We did not include the lines of code in the build system itself as a measure for complexity because a large part of the build system in modern projects is generated, rendering this metric useless for our purpose. Instead, we give the LOC of the build templates (if any) for each project.

Case study process We conduct our case study following the process in Figure 7 for each project. For every unspecified dependency that we find, we apply the following steps:

1. Manually analyze the next unspecified dependency on the list
2. Classify the dependency into an existing root cause, or add a new root cause

[6] `https://git.gnome.org/browse/glib`
[7] `http://www.openldap.org`
[8] `https://www.kernel.org/`
[9] `http://www.qt.io`

Table 1: Subject systems

	GLIB	OPENLDAP	LINUX	QT
Version	2.36.0	2.4.42	3.2	5.3.0
Build LOC	4,000	2,000	37,000	57,000
Build time	<10 mins.	<10 mins.	<20 mins.	∼4 hours
# Nodes	33,269	39,705	29,995	81,864
# Edges	270,236	164,484	2,167,450	8,440,928
strace	307 MB	218 MB	594 MB	17.5 GB
Parse time	<5 mins.	<5 mins.	<30 mins.	∼4 hours
Analysis time	<5 mins.	<5 mins.	<30 mins.	<5 mins.

Table 2: Dependencies detected in the case study

Dependencies	GLIB	OPENLDAP	LINUX	QT
# of explicitly specified dependencies	121,709	48,334	906,348	3,114,242
# of unspecified dependencies	1,657	290	944,126	284,191
% of unspecified dependencies (out of total)	1.3	0.6	51.0	8.4

3. Write a grouping operation to group dependencies with the same root cause within the list (if possible)
4. Go back to step 1, until there are no more unspecified dependencies on the list

All MAKAO and STRACE logs were collected on a dual Intel Xeon quad-core (2.53GHz) computer with 12 GB of RAM. The logs were imported and analyzed on an Intel i5 quad-core (1.70GHz) computer with 8GB of RAM. We have added the number of nodes and number of edges of the combined conceptual and concrete graph (see Section 3.3), the size of the STRACE log and the total time spent parsing and analyzing the logs with BEE for each project to the bottom half of Table 1.

In Table 2, we present the total number of unspecified dependencies that we detect in our case study. We provided the detected unspecified dependencies as part of a replication package[10]. Table 3 shows the total number of unspecified dependencies that we classify into a root cause for further analysis. We elaborate on our findings in the remainder of this section.

Manual analysis During the manual analysis phase of our case study, we semi-automatically group the unspecified dependencies into root causes. If an unspecified dependency fits into a root cause that we have already defined, we categorize it as such. If not, we added a new root cause. It is important to note that the categorization into root causes was not a subjective process. In all cases, the root cause was identified without doubt, e.g., because the root cause was explicitly specified in the source code documentation. To identify the root cause of an unspecified dependency, we followed a process that is similar to debugging, i.e., we traced each unspecified dependency in the code until the root cause was identified. This is the process we followed for every unspecified dependency:

[10] `http://sailhome.cs.queensu.ca/replication/unspecified_dependencies/`

Table 3: Unspecified dependencies per root cause uncovered in the case study

Root causes	GLIB	OPENLDAP	LINUX	QT
Generator	21	-	-	-
Meta-file	15	-	-	-
God-file	-	-	394,552	-
Guarantee	-	-	-	5,511
Helper	-	-	547,239	47,064
Temporary file	1,621	290	2,335	231,616

1. Find the rule for the generated target in the source code using `grep`. If there is no target with that name, and the file does not exist after the build, classify the root cause of the unspecified dependency as 'temporary file'. If there is no target with that name, and the file does exist after the build, classify the root cause of the unspecified dependency as 'project helper file'.
2. Search code comments, documentation or consult developers for reasons for the unspecified dependency. If an explanation is found for the unspecified dependency, classify its root cause as one of the 'project policy' root causes.
3. Otherwise, analyze the `make` code where the unspecified dependency should have been and classify the unspecified dependency into one of the other categories based on the root cause.

The first and second author independently conducted the manual classification for all four studied projects. Both manual classifications of the unspecified dependencies in the OPENLDAP, GLIB and LINUX project were identical. In the QT project, there were two small disagreements between the classifications of the unspecified dependencies. To resolve these disagreements, we contacted the QT mailing list, where one of the QT maintainers resolved the disagreements for us.

The first disagreement was about an unspecified dependency on a file that was generated during the build. The authors were unable to identify where the file was generated. The QT maintainer explained the case, after which the disagreement was easy to resolve. The second disagreement was about an unspecified dependency on a preprocessed `C++` header file while building a target that was written in `C`. The QT maintainer investigated the case and found that the file was accessed during the build by `gcc`, which read all files in a directory of preprocessed header files to identify the correct file for the language used. Because the `C++` file occurs before the `C` file in that directory, the `C++` file is accessed during the build. With the explanation of the QT maintainer, the disagreement was easy to resolve.

For every unspecified dependency that we analyzed, we also searched the list of remaining unspecified dependencies for similar unspecified dependencies and created a grouping operation for them by using commonalities in the file names of the files read or written, such as the file extension. This helped us lower the number of serious unspecified dependencies to analyze. In the next section, we will discuss our results and formalize some of these grouping operations.

5 Root Cause Analysis

During our case study, we manually classified the detected unspecified dependencies into their root causes. Below, we discuss these root causes in more detail.

5.1 Missing Generator Dependency

Description - A generator (e.g., a code robot [27]) generates code, for example, a configuration class. We classify unspecified dependencies as a missing generator dependency if a target explicitly depends on the generated code, but not on the generator itself.

Symptoms - A process p_1 that writes a target t_1 calls another process p_2 that generates code. The generated code is read by p_1 but there is no dependency from t_1 on p_2. Hence, if p_2 changes, t_1 is not rebuilt. The *generator* relation can be defined formally as:

$$generator \triangleq (reads \circ writtenBy) \cap executes$$

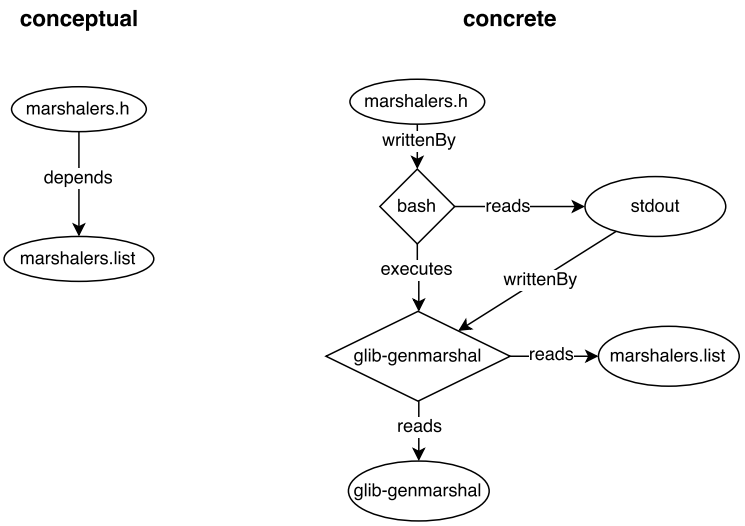
Example - Listing 3 shows an example of a missing generator dependency that we encountered in GLIB. In this example, `$(glib_genmarshal)` reads `marshalers.list` and uses it to generate code into the `marshalers.h` target. Figure 9 shows the dependency graphs. Note that there exists a *generator* relation between the `bash` and `glib-genmarshal` processes, because the `bash` process reads output that is generated by the `glib-genmarshal` process.

We want to include the files read by the generator in the set of dependencies in concrete^{A}. We can do this as follows:

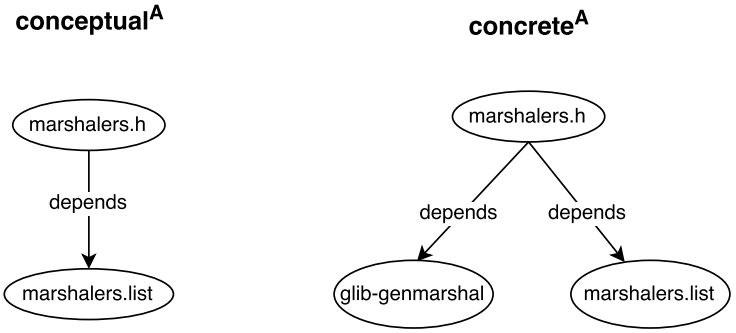
$$\begin{aligned} depends^{A} \triangleq\ & (writtenBy \circ reads) \cup \\ & (writtenBy \circ generator \circ reads) \end{aligned}$$

$G_{unspecified}$ confirms that a dependency between `marshalers.h` and `glib-genmarshal` is missing.

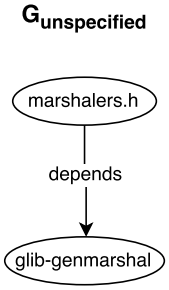
Impact - Since `marshalers.h` is missing the dependency to `glib_genmarshal`, it will not automatically be rebuilt when `glib_genmarshal` changes. Hence, `marshalers.h` may be built and shipped using outdated contents. We found 21 missing generator dependencies in GLIB. We have submitted patches for the missing generator dependencies to the GLIB project (see Section 5.6).



(a) Dependency graphs



(b) Abstracted dependency graphs



(c) Unspecified dependency graph

Fig. 9: Dependency graphs for an occurrence of a *Missing Generator Dependency* in GLIB

```
1 ...
2 marshalers.h: marshalers.list
3   $(AM_V_GEN) $(glib_genmarshal) \
4   --prefix=test $(srcdir)/marshalers.list \
5   --header --valist-marshallers > marshalers.h
6 ...
```

Listing 3: Rule that illustrates the *Missing Generator Dependency* bug in GLIB. Specifically, the right-hand side of line 2 is missing `$(glib_genmarshal)`.

5.2 Missing Meta-File Dependency

Description - A meta-file contains meta-data about a project or library, such as the data that is necessary to use a library on different platforms. Without a correctly specified dependency on a meta-file, a project may use an outdated version of a library.
Symptoms - A dependency on a meta-file, e.g., a `.la` file, is missing.
Example - GLIB uses LIBTOOL[11] for library management. LIBTOOL simplifies usage of libraries in portable projects by generating a `.la` file for a library, which contains textual meta-info. This allows projects to include the portable `.la` file instead of the platform-specific `.dll` (Windows) or `.so`/`.lo` (Linux) library. We found that while building several GLIB targets, a dependency on `libgmodule-2.0.la` is missing. Figure 10 shows the dependency graphs for `gresource` and `libgio-2.0.la`.
Impact - This type of missing dependency is a bug. We found 15 missing meta-file dependencies in GLIB. We validated that the meta-file dependencies that we found for GLIB were still missing in the latest HEAD of the source repository and we have created and submitted patches to the GLIB project (see Section 5.6).

5.3 Project Policy

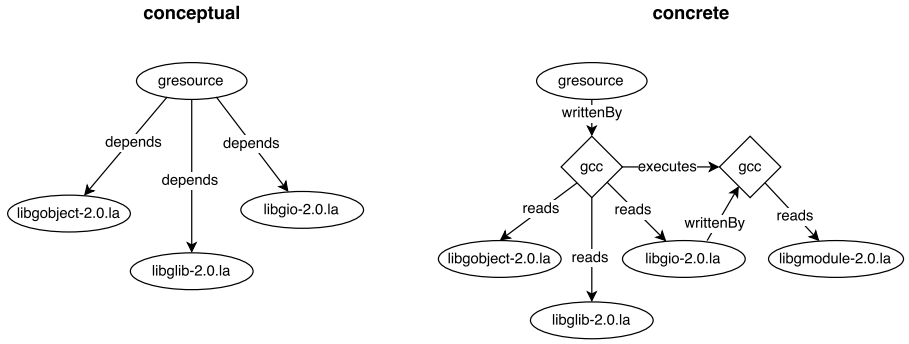
Build file incompleteness is in some projects considered to be a project policy. Reasons to keep build files incomplete can be 1) to obtain a short incremental build time, since less deliverables are being rebuilt or 2) that the expected gain of a parallelized build does not outweigh the increased number of targets being rebuilt. For example, in cases where complete build files lead to a large number of files that are unnecessarily rebuilt, parallelization may not result in shorter build times.

In some cases, project policies are controversial and subject of long discussions [17, 22]. In this section, we will discuss two of such project policies that we encountered during our case study.

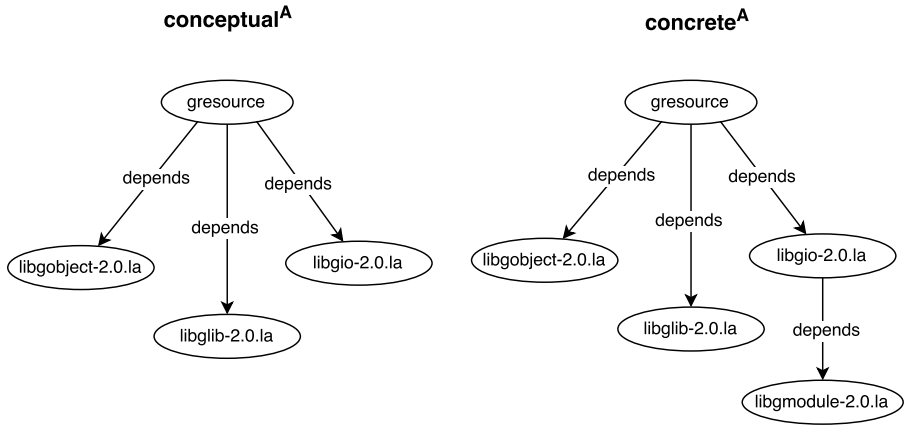
5.3.1 God-File Filtering

Description - Developers may explicitly remove certain files from the dependencies list of targets. An example of such a file is a God-file, which contains code of which

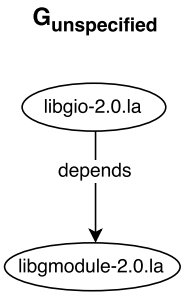
[11] `http://www.gnu.org/software/libtool/`



(a) Dependency graphs



(b) Abstracted dependency graphs



(c) Unspecified dependency graph

Fig. 10: Dependency graphs for an occurrence of a *Missing Meta-File Dependency* in GLIB

the functionality is not clearly separated. As a result, many files in the system depend on only a small fraction of the code in the God-file, causing a small change to that

file to trigger the rebuild of many targets. It is possible to automatically split the code in the God-file in separate files and create dependencies on those files instead. The source code documentation of the LINUX project describes God-file filtering in the LINUX kernel [6].

Symptoms - A dependency on a given file is unspecified in many instances.

Example - In the unspecified dependencies list for LINUX, we encounter many unspecified dependencies on `autoconf.h`. After inspection of the source code and documentation, we found that this was an optimization in the kernel build system [6]. LINUX uses `gcc` to generate a list of all the dependencies that a file has. However, `autoconf.h` is `#included` by almost every source code file. As a result, a change to `autoconf.h` triggers almost the complete system to be rebuilt. Because this file deals with the overall configuration of the kernel, most builds only use a small fraction of `autoconf.h`.

As an optimization to avoid unnecessary builds, this dependency is explicitly filtered out in the LINUX build system. Instead, the build system creates an empty file for every configuration option and adds this file as a dependency for every target that uses that configuration option in its source code. If a configuration option changes in `autoconf.h`, the build system will 'touch' that file, causing only the targets using that option to be rebuilt, rather than the whole system.

Impact - During our case study on LINUX, we found two additional cases similar to `autoconf.h` that were explicitly filtered from the dependencies, resulting in unspecified dependencies. These two cases caused 394,552 unspecified dependencies in total.

5.3.2 Binary Compatibility Guarantee

Description - A project may define a policy that forces developers to exclude some dependencies.

Symptoms - A dependency on the same file or type of file is missing in many instances.

Example - In the QT project, our approach detected a number of unspecified dependencies on `.so` (shared library) files. Those targets were depending on the header file accompanying the library. This header file serves as an interface to the library. After discussion with a QT developer, we found that QT has a binary compatibility guarantee policy [23] for patches and minor releases. A library is said to be binary compatible if a program linked dynamically to a former version of the library continues running with newer versions of the library without the need to recompile. Hence, the policy in QT states that the `.so` library may change as long as the interface to it does not change. Because the `.so` library will be dynamically linked into the target, no problem will arise. As a result, QT developers do not define the dependency on the `.so` file and only define the dependency on the corresponding header file.

Impact - We found 5,511 unspecified dependencies with the binary compatibility guarantee root cause in QT.

5.4 Project Helper Files

Description - Build systems may generate files during the full build process that are not explicitly defined in the build file, but are used during an incremental build.
Symptoms - These files are not specified as a target, but exist after the full build finishes. Removing them will result in a longer incremental build, as the build system will have to regenerate them.
Example - LINUX uses `gcc` to extract dependencies from the source code and store them in a `*.d` file during the full build, so that the `*.d` file can be used during subsequent incremental builds. Figure 11 demonstrates this. The project helper files are described by the set of files F_h:

$$helperfiles = F_h$$

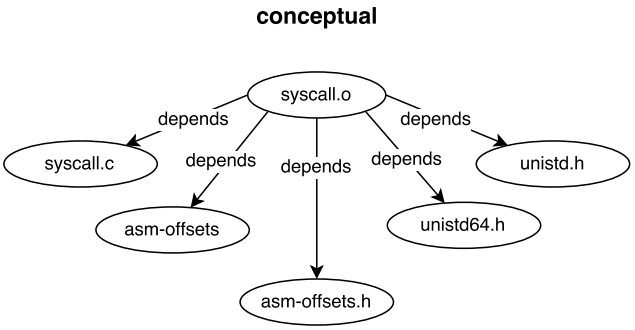
We say that $(a,b) \in depends$ if a is a helper file for building b.
Impact - We found 594,303 unspecified dependencies involving project helper files. While not directly important for build file completeness, knowing how and where such files are generated and used can be beneficial while debugging the build system. For example, if a makefile template is transformed into a makefile in several steps by a template system, it can be difficult to find out which files were used during the process.

LINUX has a high number of unspecified dependencies for project helper files because for every generated dependency file, the source code is analyzed to extract its dependencies. This causes a read operation for every (included) source file, resulting in the high number of unspecified dependencies.

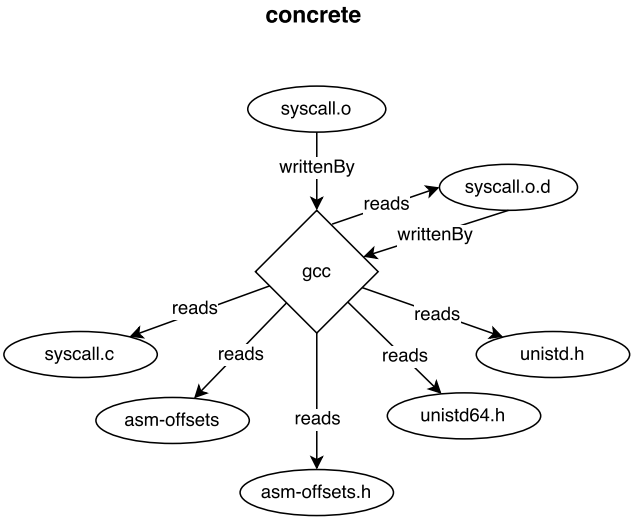
5.5 Temporary Files

Description - Build systems may generate temporary files during the full build process that are not explicitly defined in the build file and are removed before the build finishes.
Symptoms - Files that are being read by processes that write temporary files have unspecified dependencies on those temporary files. We define a file to be temporary in the same way that we define project helper files (Section 5.4), with the extra condition that the file no longer exists after the build is finished.
Example - QT generates targets in a temporary place, tests them and then moves them to their final location. As a result, there are unspecified dependencies on the temporary file because the dependencies in the build system are specified for the target in the definitive location.
Impact - We found 235,862 unspecified dependencies involving temporary files. QT has a high temporary file usage compared to the other projects because of the use of the template system, QMAKE[12] which is bundled with QT. QMAKE allows the developer to create a simple, high-level configuration file for the project instead of a

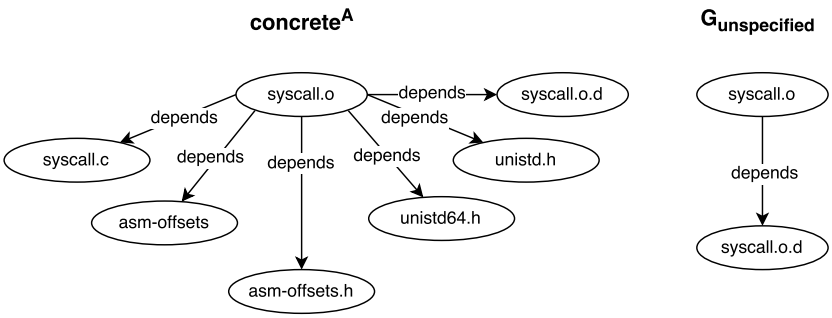
[12] `http://doc.qt.io/qt-5/qmake-manual.html`



(a) Conceptual dependency graph



(b) Concrete dependency graph



(c) Abstracted concrete dependency graph and the unspecified dependency graph

Fig. 11: Dependency graphs for an occurrence of a *Project Helper File* in LINUX

complex makefile. QMAKE automatically extracts data from the project based on that project file and generates the complex build system. In this way, the developer does not have to worry about challenging tasks, such as build system completeness and correctness. During the transformation of the configuration file into the generated build system, many temporary files are generated. Debugging QMAKE is difficult without knowing how and where these temporary files are used and generated. Hence, while dependencies on temporary files are not important for build file completeness, knowing when and where temporary files are generated can be beneficial when debugging the underlying build technology. Our approach detects such hidden information by showing the unspecified dependencies between targets and temporary files.

5.6 Feedback of Developers on the Unspecified Dependencies

We contacted a GLIB developer to validate the patches that we submitted. Although he agreed on the correctness of the patches, in this case the patches turned out not to be interesting for the project, as the situations in which the patched unspecified dependencies actually cause a bug do not occur in practice. For example, the patches that we submitted for the missing generator dependency (Section 5.1) were deemed unnecessary because the generator code was part of a compressed archive that was extracted during the build. Hence, the GLIB developer said: "it is very unlikely that the generator code is ever modified and executed without recompilation."

In addition, we spoke with a QT developer about the binary compatibility guarantee, as discussed in Section 5.3.2. The QT developer confirmed that the type of unspecified dependencies that we detected are caused by the binary compatibility guarantee.

We did not contact LINUX developers about the God-file filtering, as the LINUX kernel uses conventions (as specified in the source code documentation) to automatically optimize the build system. In addition, we did not contact OPENLDAP developers, as we only detected temporary file unspecified dependencies.

6 Limitations and Threats to Validity

In this section, we discuss the limitations and threats to validity of our case study of unspecified dependencies in `make`-based build systems.

Limitations The first limitation of our study is that we studied only the default build configuration. As our approach dynamically analyzes only one build configuration at a time, we conducted our empirical analysis only on the default build configuration for the studied projects. The limitation of being able to analyze only one build configuration at a time is inherently tied to approaches that dynamically analyze build configurations, as such approaches require to actually execute the configuration. However, as we demonstrated, static analysis is not sufficient to detect unspecified dependencies, making dynamic analysis the only option.

The second limitation of our study is that we did not calculate recall for all possible build configurations. Because our approach uses the output log as generated by

`make` and the STRACE output log of the build, it can uncover all dependencies that were not specified in the makefile for a configuration. Hence, our approach has 100% recall for the analyzed configuration. Because of the large number of possible configurations for projects such as LINUX, it is not computationally feasible to calculate the recall of our approach for all configurations in the build system. However, many of the possible configurations may rarely be used in practice. Therefore, we recommend that (1) a developer should verify the specific configuration he or she works on and (2) automatic testing should verify the most commonly-used configurations. By following these recommendations, the most important configurations are tested.

The third limitation of our study is that we studied only `make`-based build systems. In earlier work [16], GNU `make` was found to be still used by 40% of the studied repositories in 2012. Our approach is independent of the build technology as long as it is possible to (a) create a conceptual dependency graph from build file-level information and (b) create a concrete dependency graph from execution-level information. For our implementation, we concentrated on makefiles for two reasons: (1) they are widely used (as described in Section 2.1, many higher-level build technologies such as `CMake`, and `Autotools` generate makefiles), and (2) we were able to reuse tools (e.g., MAKAO) that were already implemented to deal with makefiles and their execution tracing. Because our implementation works with `make`-based technology, it does not work with build technologies that are not `make`-based, such as Google Bazel.

External Validity We have conducted our case study on the build systems of four open source projects. These projects are widely used and maintained by a mature community, making it more likely that 'low-hanging fruit' of the problematic unspecified dependencies in the build system have been identified already. Even so, our approach is still able to uncover a large number of unspecified dependencies with several different root causes, including root causes that can be classified as bugs. Our approach is semi-automated, making it easy to repeat our case study on other systems.

We acknowledge that we did not study unspecified dependencies in the build systems of industrial systems. In future work, we will perform a large study in which more open source and industrial projects are analyzed.

Construct Validity Build systems contain build information for many targets. In our evaluation, we used the default build target of the build systems. This target builds the most commonly used targets in the build system, i.e., the targets that produce the executable used by most people. In addition, these targets are less likely to contain unspecified dependencies because they have been executed and analyzed many times before. While we acknowledge that building the default target does not necessarily build all targets in the system, we opted to analyze only the most commonly used.

Note that we did not calculate the interrater agreement score for our manual analysis. In our study, we followed a consensus-driven approach to explain unspecified dependencies, i.e., we discussed disagreements and arrived at an agreement for every unspecified dependency. Therefore, as we do not classify the unspecified dependencies into predefined classes, interrater agreement scores such as Cohen's kappa [3] are not applicable to our study.

Internal Validity A possible threat to the internal validity of our empirical analysis is that we only investigated projects that use a build system based on `make` during our case study. The reason for this is that our available tool set was designed for `make` build systems. As `make` is a mature and widely used build system technology, we do not see this as a threat to the validity of our case study results.

7 Discussion

Our method was very successful at finding unspecified dependencies (we found over 1.2 million unspecified dependencies). However, we were surprised that most of these missing dependencies were condoned by the development teams, i.e., they knew about the existence of the unspecified dependencies, but chose not to fix them. The major reason for not fixing those unspecified dependencies is to avoid large incremental builds; hence, the developers consider (rightly or not) that touching these dependencies should not trigger rebuilds.

While investigating these unspecified dependencies, we discovered that they are managed in an ad-hoc manner, meaning there was no standard approach for omitting dependencies, but instead, developers implement custom methods for this. These custom methods are probably necessary because build tools do not consider this use-case; however, as demonstrated in the empirical study, there is a need to support this feature.

The systems that we analyzed are old and mature. Their makefiles, and the dependency specifications they contain, have been tested during a long period. It is not surprising that the developers of the studied projects are mostly aware of the unspecified dependencies that we uncovered and that we found only 36 actual missing dependencies (in GLIB). Because the build systems of the studied projects have been thoroughly tested over the years, it is unlikely that we would uncover bugs which are easily detected manually. However, even for an old project such as GLIB, we were able to uncover unspecified dependencies that could potentially lead to bugs in the future. The fact that we were able to uncover unspecified dependencies that are potentially dangerous in such an old project, indicates that our approach is useful for improving a build system when it becomes difficult to improve manually.

We believe that dependency analysis can be especially beneficial when files (especially source code files) are added to a project. One can verify, as part of continuous integration, that if the new file is present in the concrete dependency graph, then it should also be present in the conceptual dependency graph.

It is also important to recognize that there are many different potential concrete dependency graphs when building a software system. For example, LINUX has hundreds of configuration switches that include/exclude features from a build. The number of potential concrete dependency graphs for LINUX is huge. Any developer who works in a given configuration should, before making changes, verify that the build specification has no unspecified dependencies, especially of the files she is expecting to modify. The method that we propose herein can be useful in such situations.

The Tarski formulas create the theoretical foundation for the implementation of our prototype. Hence, we consider each (the model described using Tarski formulas

and our prototype) as one of the two important contributions of our paper. The Tarski formulas formalize how the conceptual graph (resulting from analysis of makefile-related data by MAKAO) and the concrete graph (resulting from analysis of the actual build by Bee) are combined into a single graph and how this graph should be queried to identify unspecified dependencies. We make an empirical contribution by implementing a prototype that is built upon this model. Both Bee and MAKAO model the build process as a graph, and we combine these graphs and query them using the model that is built upon the Tarski formulas.

8 Conclusion

Unspecified dependencies are dependencies that are used during the build process while they are not explicitly specified in a build file. Such unspecified dependencies can cause inconsistencies or bugs in targets, which are generated by `make`-based build systems that are executed recursively or in parallel.

In this paper, we presented the results of our empirical study on unspecified dependencies in the `make`-based build systems of four open source projects (GLIB, OPENLDAP, LINUX and QT). Our analysis revealed over 1.2 million unspecified dependencies caused by 6 root causes. While some of these root causes are actual bugs, most are known by the developers of the studied projects. In this paper, we have made the following contributions:

- A case study on the build systems of four open source projects in which we manually validate the results of our approach
- A discussion of the 6 root causes of 1.2 million detected unspecified dependencies
- The detection of 36 instances of bugs with 2 root causes in the mature build system of GLIB

Our paper shows that, even though there exist tools to avoid unspecified dependencies (e.g. memoize), unspecified dependencies are real and exist in popular open source projects. In fact, even in a mature project such as GLIB, we could detect unspecified dependencies (even though the situations in which these dependencies could lead to bugs are rare). Our findings show that it is important to search for unspecified dependencies in all projects – even older ones.

References

1. Adams B, Tromp H, De Schutter K, De Meuter W (2007) Design recovery and maintenance of build systems. In: IEEE International Conference on Software Maintenance (ICSM), pp 114–123
2. Adams B, De Schutter K, Tromp H, De Meuter W (2008) The evolution of the linux build system. Electronic Communications of the EASST 8
3. Banerjee M, Capozzoli M, McSweeney L, Sinha D (1999) Beyond kappa: A review of interrater agreement measures. Canadian Journal of Statistics 27(1):3–23, DOI 10.2307/3315487, URL `http://dx.doi.org/10.2307/3315487`

4. Coetzee D, Bhaskar A, Necula G (2011) apmake: A reliable parallel build manager. In: 2011 USENIX Annual Technical Conference (USENIX)
5. Feldman SI (1979) Make – a program for maintaining computer programs. Software: Practice and experience 9(4):255–265
6. Germaschewski K (2002) fixdep.c Source Code. https://git.kernel.org/cgit/linux/kernel/git/stable/linux-stable.git/tree/scripts/basic/fixdep.c?id=refs/tags/v3.2.71, [Online; accessed December 21, 2016]
7. Google (2015) Bazel. http://www.bazel.io/, [Online; accessed December 21, 2016]
8. Google (2015) Bazel: Concepts and terminology. http://www.bazel.io/docs/build-ref.html#actual_and_declared_dependencies, [Online; accessed December 21, 2016]
9. Gunter CA (2000) Abstracting dependencies between software configuration items. ACM Trans Softw Eng Methodol 9(1):94–131
10. Hassan A, Zhang K (2006) Using decision trees to predict the certification result of a build. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 189–198
11. Hoyt B (2009) Fabricate - the better build tool. https://code.google.com/archive/p/fabricate/, [Online; accessed December 21, 2016]
12. Jørgensen N (2002) Safeness of make-based incremental recompilation. In: FME 2002:Formal Methods – Getting IT Right, Lecture Notes in Computer Science, vol 2391, Springer Berlin Heidelberg, pp 126–145
13. Leslie B (2008) Memoize: a build tool framework. http://benno.id.au/blog/2008/06/06/memoize-build-framework, [Online; accessed December 21, 2016]
14. McIntosh S, Adams B, Nguyen TH, Kamei Y, Hassan AE (2011) An empirical study of build maintenance effort. In: Proceedings of the 33rd International Conference on Software Engineering (ICSE), ACM, New York, NY, USA, pp 141–150
15. McIntosh S, Adams B, Nagappan M, Hassan A (2014) Mining co-change information to understand when build changes are necessary. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp 241–250
16. McIntosh S, Nagappan M, Adams B, Mockus A, Hassan A (2014) A large-scale empirical study of the relationship between build technology and build maintenance. Empirical Software Engineering pp 1–47
17. Michael Elizabeth Chastain (1999) Re: corrupt modversions.h built in 2.2.13. https://lkml.org/lkml/1999/10/22/172, [Online; accessed December 21, 2016]
18. Miller P (1998) Recursive make considered harmful. AUUGN Journal of AUUG Inc 19(1):14–25
19. Nadi S, Holt R (2011) Make it or break it: Mining anomalies from linux kbuild. In: 18th Working Conference on Reverse Engineering (WCRE), pp 315–324
20. Nadi S, Holt R (2012) Mining kbuild to detect variability anomalies in linux. In: 16th European Conference on Software Maintenance and Reengineering (CSMR), pp 107–116

21. Neitsch A, Wong K, Godfrey M (2012) Build system issues in multilanguage software. In: 28th IEEE International Conference on Software Maintenance (ICSM), pp 140–149
22. OCaml Community (2013) Dependencies are missing in makefiles - MantisBT. http://caml.inria.fr/mantis/view.php?id=4243, [Online; accessed December 21, 2016]
23. Qt Wiki (2015) Qt-Version-Compatibility. https://wiki.qt.io/Qt-Version-Compatibility#Binary_Compatibility_Guidelines, [Online; accessed December 21, 2016]
24. Seo H, Sadowski C, Elbaum S, Aftandilian E, Bowdidge R (2014) Programmers' build errors: A case study (at google). In: Proceedings of the 36th International Conference on Software Engineering (ICSE), ACM, New York, NY, USA, pp 724–734
25. Tamrawi A, Nguyen HA, Nguyen HV, Nguyen T (2012) Build code analysis with symbolic evaluation. In: 34th International Conference on Software Engineering (ICSE), pp 650–660
26. Tarski A (1941) On the calculus of relations. The Journal of Symbolic Logic 6(03):73–89
27. Tu Q, Godfrey MW (2001) The build-time software architecture view. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM), IEEE Computer Society, Washington, DC, USA, pp 398–
28. Xia X, Lo D, Wang X, Zhou B (2014) Build system analysis with link prediction. In: Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC), ACM, New York, NY, USA, pp 1184–1186
29. Yamada M (2015) [PATCH v2] ARM: add boot image dependencies to not generate invalid images. https://groups.google.com/forum/?fromgroups#!topic/fa.linux.kernel/00v6OF3KaSM, [Online; accessed December 21, 2016]
30. Zhou B, Xia X, Lo D, Wang X (2014) Build predictor: More accurate missed dependency prediction in build configuration files. In: IEEE 38th Annual Computer Software and Applications Conference (COMPSAC), pp 53–58