The Forgotten Case of the Dependency Bugs

On the Example of the Robot Operating System

Anders Fischer-Nielsen
SQUARE Group, IT University of Copenhagen

Zhoulai Fu
SQUARE Group, IT University of Copenhagen

Ting Su
ETH Zurich

Andrzej Wąsowski
SQUARE Group, IT University of Copenhagen

ABSTRACT

A dependency bug is a software fault that manifests itself when accessing an unavailable asset. Dependency bugs are pervasive and we all hate them. This paper presents a case study of dependency bugs in the Robot Operating System (ROS), applying mixed methods: a *qualitative* investigation of 78 dependency bug reports, a *quantitative* analysis of 1354 ROS bug reports against 19553 reports in the top 30 GitHub projects, and a *design* of three dependency linters evaluated on 406 ROS packages.

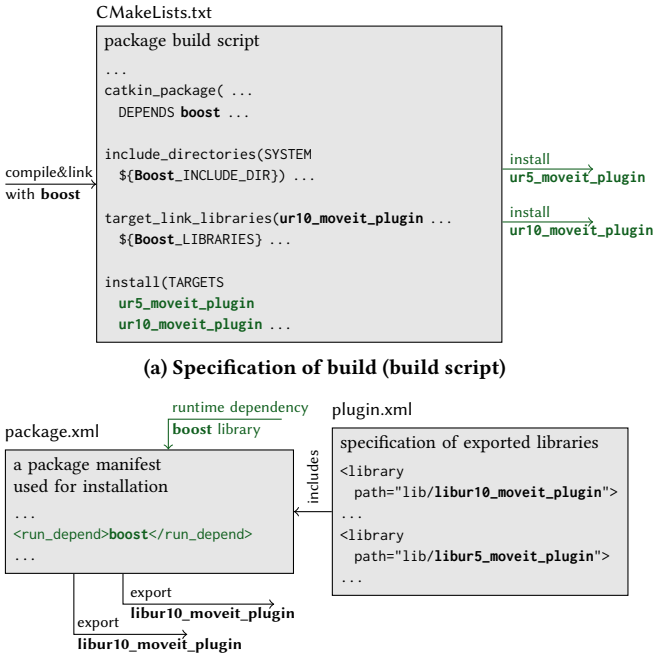
The paper presents a definition and a taxonomy of dependency bugs extracted from data. It describes multiple facets of these bugs and estimates that as many as 15% (!) of all reported bugs are dependency bugs. We show that lightweight tools can find dependency bugs efficiently, although it is challenging to decide which tools to build and difficult to build general tools. We present the research problem to the community, and posit that it should be feasible to eradicate it from software development practice.

ACM Reference Format:
Anders Fischer-Nielsen, Zhoulai Fu, Ting Su, and Andrzej Wąsowski. . The Forgotten Case of the Dependency Bugs: On the Example of the Robot Operating System. In . ACM, New York, NY, USA, 10 pages.

1 INTRODUCTION

For years, research on bug finding and fixing has prioritized rare, but intricate, bugs that are hard to avoid for programmers. This includes memory management bugs (leaks, use-after-free, null pointer dereferences, buffer overruns), concurrency (deadlocks, live-locks, and data races), typing (wrong casts, wrongly composed expressions), data flow (uninitialized variables), floating point errors (overflows, underflows), etc. Here, we investigate another category of bugs, which might not be considered worthy by some, but can be more frequent than expected: the *dependency bugs*. These bugs encompass a large group of structural problems manifested by an asset (such as a data file, a code module, a library or an executable file) being unavailable when needed, at compile-time, test-time, initialization-time, run-time, or deployment-time.

Dependency bugs appear because of modular construction of software, use of multiple languages, and independent evolution of components and languages. Dependency bugs are particularly painful for programmers new to a project. They often lack intricate, but otherwise common, knowledge to avoid them, and struggle to fix them, which shows in many online posts and discussions about simple dependency errors. Senior developers in large projects also suffer from dependency bugs. For them the size and the independent evolution of multiple parts of a system appear to be a reason for mis-specifying dependencies.



(a) Specification of build (build script)

(b) Specification of packaging (manifest)

Figure 1: A dependency bug in ROS Universal Robot driver

In this paper, we study dependency problems in Robot Operating System (ROS).[1] ROS is a platform for building robotics systems, using multiple languages, a custom advanced build system (based on CMake), and its own package management system. Figure 1 presents an example of a past dependency bug[2] extracted from the history of ROS, more precisely of an open source driver for a popular family of robot arms by Universal Robots. The fault lies in inconsistency of specification of dependencies between the build script and the packaging manifest for a module. The green elements are missing in the original code, and have been added in the bug-fixing patch. The top diagram, Fig. 1a, shows a fragment of a CMake build script that depends on the boost library for compilation (include directories) and linking (target link libraries). The bottom Fig. 1b shows a fragment of the manifest, a so called `package.xml`,

[1] http://www.ros.org
[2] Issue: https://github.com/ros-industrial/universal_robot/issues/113, fixing commit: https://github.com/fmessmer/universal_robot/commit/e50a44, ROBUST entry: https://github.com/robust-rosin/robust/blob/master/universal_robot/040fd11/040fd11.bug

,
,
.

that defines how the software interacts with the package manager. A `package.xml` manifest defines both what code assets need to be installed to compile, link, test, and package the code, and what dependencies must be available at runtime on user's platform. As shown in the figure, while the build script (top) depends on the boost library, the runtime part of the manifest does not require that this library is available at runtime. This may cause compilation errors for the user (boost header files missing) or crashes at dynamic linking time (for second level users who installed the binaries only); situations especially confusing for newcomers to ROS and C++.

Furthermore, the manifest (Fig. 1b) includes specification of exported libraries for two sizes of robot arms, ur5 and ur10. These are also built in the `CMakeLists.txt` script above, but *they are not installed*. Even though `package.xml` exports them to users, they will not be included in the binary package, and will cause failures of compilation or dynamic linking in dependent packages. This issue has been fixed by adding two new install targets to the build script (`ur5_moveit_plugin` and `ur10_moveit_plugin`, green in the figure).

The above example is fairly representative among dependency bugs in ROS. It is reasonably simple to debug and fix for a programmer who navigates the project infrastructure well. At the same time, it appears convoluted and complicated to others, especially relative to the conceptual simplicity of the problem (missing files).

Since 2017, we have worked on a participatory action research project, ROSIN, with the ROS community.[3] Within ROSIN, approximately 200 historical bugs in ROS have been collected and analyzed.[4] We were surprised by the challenges that dependency bugs pose in daily work. It was perplexing to see that 10% of the surveyed bugs were dependency bugs [24] (weakly representative sampling). We realized that dependency bugs may be a significant productivity problem, and a detractor for potential community members. We decided to investigate the issue closer, in a dedicated set of experiments, asking the following research questions (RQ):

RQ1. *What are dependency bugs? What kinds exist?*
RQ2. *How pervasive and difficult to handle are dependency bugs?*
RQ3. *Can lightweight strategies help finding dependency bugs?*

We explored these questions in a mixed methods case study in ROS. We analyzed 78 dependency bugs in ROS qualitatively to derive a definition of dependency bugs and a taxonomy of their kinds. We used quantitative approximation of the problem, to estimate how common it is among 406 ROS packages, 1354 ROS bug reports, and 19553 bug reports in 30 top GitHub projects. Finally, we prototyped three dependency linters, to experience the challenges these bugs present to builders of bug checkers. We found that:

- Most common problems are missing build-time and run-time dependencies. More complex cases include dependencies not correctly exported (like above) or circular, spurious dependences causing package conflicts, and dependency links being wrongly calculated. Sometimes, bugs appear even if neither the dependant and the depending party has changed, instigated by changes in the package distribution, the operating system, or the programming language.
- Dependency bugs are frequent: they appear with similar frequency in ROS and in top 30 GitHub projects (ca. 15% of all issue reports), they can be found in about 53% of ROS projects, and ca. 30% of contributors are involved in discussing them.
- Dependency linters can be build using little code (in the order of 100 lines) for specific cases. It is difficult though to build reusable dependency tools as most bugs are idiosyncratic to the platform. Also, it is hard to understand which dependency bugs are likely in a project (without an extensive analysis), which hampers prioritizing particular linters.

We hope that the results presented in this paper will get the attention of the community, and help to realize that dependency bugs are a significant problem and annoyance, and that they cannot be eliminated solely by continued program analysis research that tends to be enclosed in the universe of a single programming language silo. These problems are sufficiently annoying to warrant eradicating them, and sufficiently simple to give us hope that this is feasible.

2 ROBOT OPERATING SYSTEM

ROS is the most popular open-source platform for robotics. It provides standard communication and coordination features (distributed middleware), along with essential robotics-specific components, such as drivers for popular hardware and modules for control, perception, localization, etc. A ROS-based application is created by selecting and configuring the necessary components, and complementing them with application specific code, typically in Python or C++ [19]. ROS is widely used in teaching and research. For example, MoveIt!, a planner for ROS, has been used to control more than 65 kinds of robots [3]. Recently ROS attracts increasing industrial interest. The ROS-Industrial consortium counts over 70 members.

ROS has its own CMake-based build system (catkin for ROS1, ament for ROS2) and its own package manager (rosdep) that interfaces to the underlying platform (chiefly via apt-get and pip) to construct the build-time and runtime environment for ROS packages. A deployment configuration mechanism, roslaunch, interprets ROS-specific startup scripts that define which nodes to start, how they should communicate, and what parameters values to set. With this design ROS projects are highly modularized, and ROS programmers use many languages, starting from C++, through python, to several DSLs (`package.xml`, launch scripts, robot definition models, message definitions, etc.). ROS packages are distributed in collections known as distributions, temporally aligned with Ubuntu distributions. (Ubuntu Linux is the main platform underlying ROS.)

3 RQ1: DEPENDENCY BUGS

We present the exploratory qualitative inquiry into dependency bugs. The objective is to characterize and define dependency bugs.

3.1 Method

We worked in two modes alternately: (i) an *analytical* mode (Section 3.2), where we analyzed and categorized dependency bugs; and (ii) a *synthesis* mode (Section 3.3), where we evolved the definition from analyzed cases. Our data include pull requests and issue

[3] https://rosin-project.eu/
[4] The data is not finalized yet, but the process of collecting it, including the current snapshot is publicly available at: https://github.com/robust-rosin/robust

	Description	#	Location	Dependency Fault	Failure Stage
1	**Build-time dependency missing**	**33**		missing	build
2	in build script	8	build script	missing	build
3	in manifest	18	manifest	missing	build
4	missing linker dependency (build not linking against the required library)	2	build script	missing	build
5	build-tool dependency missing	5	manifest	missing	build
6	**Run-time dependency missing**	**23**			
7	runtime dependency specification missing in the package manifest	23	manifest	missing	run
8	**Test- time dependency missing**	**1**	manifest	missing	test
9	**Wrong build-time dependency**	**13**			build
10	linking against multiple versions of the same library	4	build script	inconsistent	build
11	dependency on wrong programming language version (compiler option)	1	build script	wrong	build
12	build-time dependency not passed down to dependent packages	7	build script	not exported	build
13	build-time dependency is unconditional, but should be conditional	1	build script	wrong	build
14	**Spurious (unused or otherwise not needed) dependency**	**12**			
15	in build script	6	build script	spurious	build
16	in manifest	6	manifest	spurious	build/run/test
17	**Circular dependency (in various dependency graphs)**	**1**	various	circular (wrong)	build
18	**File path problems at various phases**	**8**			
19	wrong file path computation in dependencies	4	build script/manifest	wrong	build/run/test
20	wrong location for compiled artifacts	1	build script	wrong	build
21	wrong executable name in a launch (start) script	2	launch script	wrong	launch/test
22	wrong location for install files	1	build script	wrong	installation
23	**Operating system/platform/distribution issues**	**7**			
24	wrong operating system version	2	other	wrong	build/run/test
25	cross platform incompatibility between resources	2	build script/manifest/code	missing	build/run/test
26	dependency not released	3	manifest	missing	installation
27	**Source code issues**				
28	wrong module import in source code	**2**	code	wrong	build/run
29	**Meta-dependency problems (NOT dependency problems)**	**7**	dep. mgmt. mechanism	–	build/run/test

Table 1: Classification of dependency bugs in ROS Melodic (some issues contain more than one bug, so # does not sum to 78)

reports collected from 455 packages in the ROS Melodic Morenia distribution.[5] We performed four iterations:

(1) We picked nine issue candidates of GitHub issues from three popular ROS repositories: catkin, py_trees, and turtlebot3. Each issue contains the word 'dependency' or 'dependencies.' We analyzed, discussed, and labeled the candidates that we believed to be dependency bugs, based on the common informal understanding what is a dependency bug (like in Section 1). We captured a short rationale for each classification decision (the analysis mode). This discussion led to the first formulation of a definition of a dependency bug (the synthesis mode).
(2) We searched for another 20 issue tracker entries with the substring 'depend.' We discussed them in a joint meeting (the analysis mode), finding that the keyword search worked reasonably well for the purpose of qualitative analysis. Then, we refined the definition (the synthesis mode) and automated the search process with a script. The script checked for presence of the substring 'depend' (shortened to cover lexical variations) in the title, body, and all discussion comments of an issue tracker entry. The script identified an issue as a bug if developers tagged the issue with any label containing the term 'bug', for example 'bug', 'type: bug' or 'todo: bug.' We were inspired by Vasilescu and coauthors to use this approximation entries representing bug reports and bug fixes [25]. At the point of the experiment the distribution included 455 repositories, out of which 118 had at least one issue tracker entry tagged as a bug.
(3) Using the script, we collected new 100 GitHub issues, 50 including 'depend' and 50 not including. We used GHTorrent [9] to obtain these data. We randomly distributed them to the paper authors, withholding the script labeling from them. Each author labeled the issues manually, capturing a short rationale. We discussed all nontrivial cases in a joint meeting, including the dependency bugs that tested negatively for the keyword inclusion, and non-dependency bugs that tested positive for 'depend' (the analysis mode). We proposed a refined definition as the result of this iteration (the synthesis mode).
(4) Finally, we used the script again to gather all the remaining 95 bugs in the ROS Melodic distribution that tested positive for inclusion of 'depend' and 'bug'. We classified them manually, flagging interesting cases and discussing them (analysis). At this stage, no change to the main definition was necessary, thus we concluded saturation (synthesis).

In total, we manually classified 224 issue tracker entries and labeled 78 of them as dependency bugs. (The distribution contained 25596 issues and pull requests, where 1354 were tagged 'bug'.) We used a three valued classification, asking: Is this a dependency bug? With possible answers: Yes, No, and Invalid. The latter was used for entries that contained insufficient amount of information to conclude, and for the few entries incorrectly tagged 'bug' by developers. The main outcome of this process is the definition of a dependency bugs presented in Section 3.3. For simplicity, we only show the final version of the definition. To reflect the process, Section 3.2 presents the

[5] https://github.com/ros/rosdistro/tree/master/melodic, captured on 2019/06/22

various kinds of dependency bugs that we have seen in ROS, demonstrating how rich this space is, and giving rise to the first known taxonomy of dependency bugs extracted from a real system data.

3.2 Analysis: The Space of Dependency Bugs

To understand the space of dependency bugs we have manually coded the bugs by the nature of the failure and the fault. We arrived at 21 codes for dependency bugs, and one not representing a dependency bug but a related issue. The codes have been manually clustered into ten exclusive groups by conceptual proximity. We describe the spectrum of dependency problems in ROS using these groups. For each group we link to an example, so that the reader can confront our observations with the source material on GitHub.

Table 1 summarizes our sample of bugs, organized by codes. It lists the groups, followed by number of occurrences in the sample, the location of the fault and the manifestation of the failure. The few gray entries represent variants that we believe are possible and reasonably likely, but have not been found in the sample. The reader is invited to refer to the table throughout the discussion.

Build-time dependency missing.[6]

Fault The problems in this group are caused by missing dependency declarations in either a package manifest or a build script. The missing dependencies may refer to other source packages, binary libraries, and additional tools used in the build process. Most often, the code necessary for building is not installed pre-build time. Sometimes, the pre-requisites are installed but the build script does not make them visible for the build process (for instance not putting them in the right location, or in the correct environment variable). The fault may also be located in the source code: for instance the build script correctly declares no dependency, but the program uses a component needlessly, or against guidelines. In such case, the bug is fixed by removing offending code, instead of adding the dependency. This happens relatively rarely though.

Failure Bugs in this group result in a build process failure, either during the preparation time, compilation (missing headers), or linking (unresolved symbols). These problems are detected statically by tools, but the results are brittle, dependent on the configuration of the machine on which the check is made. Some survive checks even in the pristine environment of continuous integration (CI).

Observation 1. *Despite that tools find build bugs automatically and effectively, these bugs have not disappeared from committed code. Missing build-time dependencies are the most frequently reported, discussed, and fixed group of dependency bugs in ROS.*

We might be seeing so many of these exactly because static dependency checking *is* successful. Build tools and package managers check for these errors, to prevent others, like runtime problems.

Runtime dependency missing[6]*.*

Fault A missing runtime dependency is a bug in a package manifest. When a runtime dependency is missing, the binary package built from the manifest is not announcing a required prerequisite. The prerequisite will not be available when the package is used.

Failure These bugs result in failures at runtime, typically at startup time (missing dynamically linked libraries, helper scripts, and tools), but sometimes later (for instance if a dependency is only rarely accessed in Python, which does not require linking at load time). Both with build-time and runtime dependencies missing, the original developer would often miss the problem, because the prerequisites are installed in his work environment for other reasons.

Missing runtime dependencies in ROS occur almost always together with a missing build dependency in a manifest file and in a build script. This appears to be a reflection of code-oriented work flow of developers, where an asset is first used in the code, and then specified as a dependency. This last step is forgotten because the dependency is already installed on the developer's machine.

Observation 2. *Redundancy of dependency specifications is contributing to introducing dependency bugs, or even causing them. Similar dependency information needs to be specified both in the build script, in the package manifest, and in the source code.*

Dependencies play slightly different roles in these places, and there exist use cases, where these three are not identical on purpose, or when not all three are used. Possibly though, some of the redundancy is caused by the fact that the various tools dealing with dependencies are developed independently, by different projects. Reducing or elmiinating redundancy of specifications would require further design work and collaboration between various tools.

Test-time dependency missing.[7]

Fault Modern build systems and package management systems (including catking and rosdep) distinguish runtime dependencies from test dependencies. The former are meant to be imposed on all users of the software, the latter only on developers and CI.

Failure A missing test dependency usually results in the tests crashing (e.g. due to failure of dynamic linking), throwing exceptions (missing python packages), or otherwise terminating.

Wrong build-time dependency.[8]

Fault This more subtle category concerns bugs where build dependencies are specified, but they are flawed. The most common flaw in ROS is that dependencies are not exported during construction of a binary package (post-build installation). We also observed problems like linking different parts of a system with different versions of the same binary library, depending on a wrong version of programming language (in ROS these is either using wrong version of Python, or C++, both with non-trivial differences between versions), and using an unconditional dependency, where a conditional should be used.

Failure Since broken dependencies have many natures, they can have various consequences. Most commonly though, they cause a missing build dependency in another package. If a build script does not export a dependency to the binary package, then it will not be installed by dependent packages. Programmers building against such binary package will experience missing header files, etc.

[6]Example: https://github.com/ros-drivers/openni_camera/issues/21, both a runtime and build-time dependency problem. Note that the bug in Fig. 1 is also a missing runtime dependency problem (the boost library is not listed)

[7]Example: https://github.com/ros/ros/pull/181

[8]Example: https://github.com/ros/geometry/issues/21. Note that the example of Fig. 1 is also a case of a wrong build-time dependency, where the plugins are exported but not installed post-build.

Observation 3. *As dependencies are a phenomenon relating two ends, the dependency and the dependant, they often cause problems on both sides. This makes detecting dependency bugs cumbersome (requires cross-package work), but also raises opportunities to more precise diagnosis and repair. Information coming from the other end of the dependency may help to disambiguate problems.*

Spurious dependency.[9]

Fault Spurious dependencies are dependencies on assets that are not used in the source code, do not exist, or can be forced to exist in other ways, without explicit specification. They appear when functionality is removed, when packages are merged or otherwise refactored, when dependencies are exported from another package or hard-coded tool in the management system.

Failure Some spurious dependencies tend to be just bad smells—they make installation harder for users (more prerequisites to satisfy) and needlessly increase chances of name and library–version conflicts. Yet, they do not cause immediate failures. Others create problems immediately; For instance, if a dependency has been removed from a distribution, it cannot be installed and the build fails.

Circular dependency.[10]

Fault Most dependency management systems require that the global dependency graph is acyclic. Occasionally it happens that this constraint is violated by developers.

Failure In such case, the users of the dependency management mechanism experience a mechanism failure at the time of attempted installation or build (a dependency check). The system is usually unable to point out the offending component, as the cycle is inherently a global issue. Unless caused by simple mistakes, these problems may require rearchitecting the offending part of the system.

File path problems.[11]

Fault Many assets on which ROS packages depend are stored in files. Dependencies are often broken because the paths of the files are wrong. This can be cause by a mistake in a computation (if the path is computed), by artifacts being located in a wrong place (while the path is correct), and by names being misspelled.

Failure File path problems manifest themselves by failures in all stages of dependency management: at build, packaging, installation, and run-time, depending on the role of the referenced artifact.

Operating system/platform/distribution issues.[12]

Fault Several dependency issues in the sample have been caused by developers not testing code on all used platforms, a common challenge for cross-platform systems. Here, we include also problems when distributions and programming languages do not offer the same set of packages as used by the developer, mostly because binary distributions are behind the source repositories.

Failure Most OS- and platform-induced dependency problems manifest similarly to missing dependency specifications. Typically a component is either not available for the given platform, or not yet released, resulting in a failure of build or installation.

[9]Example: https://github.com/jsk-ros-pkg/jsk_recognition/issues/2087
[10]Example: https://github.com/ros-simulation/gazebo_ros_pkgs/issues/24
[11]Example: https://github.com/ros-drivers/joystick_drivers/issues/30
[12]Example: https://github.com/splintered-reality/py_trees/issues/83

Observation 4. *The issues with dependency specifications changing at various speed in databases, and operating systems differing with what dependencies they can satisfy, demonstrate particularly clearly that dependency problem detection is not a standard program analysis problem. In here, the primary challenge for bug prevention tools is not the complexity and decidability of the analysis algorithm, but access to all required dependency information, possibly across platforms and temporal shifts. The challenge is further exacerbated by the fact that the various parts of the ecosystem are under control of various persons, and it is not always clear on which side the issue should be fixed.*

Source code issues.

Fault Not all dependency issues are placed in dependency specifications (manifest files and build scripts). Programming languages have dependency management mechanisms as well, mostly via import statements for packages and name spaces. Especially in dynamically linked (C++) and dynamic (Python) languages these interact with dependency specifications—imported packages must be provided, usually via the outer dependency mechanism. These bugs are like missing dependency bugs, but the offending side is the code in programming language not the manifest files and scripts.

Failure For all these errors, the symptoms are the same as for wrong and missing dependency bugs. The difference is in cause.

Programming language dependency bugs are typically detected by name analysis, type system, or a linker. So why do they exist in committed code? The language infrastructure can only check these bugs in a particular build environment (the developer's machine).

Observation 5. *The programming language tools lack information about how the build environment has been constructed, and they are unable to assess, whether the build or launch of the package on another machine will also succeed without errors.*

Meta-dependency problems.[13]

Fault A related class of bugs are faults and shortcomings in the dependency management software itself, so bugs in build systems and package managers. These bugs are classic software bugs, unrelated to dependencies, that occur in the logics resolving dependencies. We call them *meta-dependency* bugs. They are usually *not* dependency bugs per sé. They require very different methods to detect them and correct (in particular, they usually cannot be detected by the dependency mechanism used by the dependency software).

Failure These bugs are deceivingly similar to dependency bugs, as they can easily result in builds and installations failing and in dependencies not being found in some concrete packages using the build system, compiler, linker, or package manager. This can be very confusing for users, as they would typically suspect that the fault lies with their dependency specification and not with the tool.

3.3 Synthesis: Definition of Dependency Bugs

The reader who followed the example links in the previous section will appreciate how much accidental complexity dependency bugs present. One objective for this work is to synthesize a catalog describing the space of these bugs (above) and a common definition.

[13]Example: https://github.com/ros/catkin/issues/867

We believe that elucidating the problem is the first step in finding a good solution. We have analyzed the data set described above and proposed a definition of dependency bugs:

Definition 3.1. A *dependency bug* is a software fault that both

(1) Manifests itself when accessing a software asset that is unavailable (either because it does not exist, is located elsewhere than anticipated, or it is broken)
(2) Can be fixed by modifying configuration specification or code to adjust (change, insert, remove) asset names and references.

An *asset* should be understood broadly as anything that can be referenced, and is provided externally. It could be a software package, a module, a class, but also a script, an external tool, a library, an operating system, a database, a service, a file, or a specification of dependencies itself. A *configuration specification* could be a package manifest, a build script, a distribution index, a helper script, or a file defining options and deployment. The dependency bugs manifest themselves in any life cycle phase of a system, including the pre-build installation (constructing the workspace), build-time (compilation and linking), test, static analysis, installation, deployment and packaging (post-build installation), and at runtime.

4 RQ2: DEPENDENCY BUGS CHALLENGE

We now attempt to estimate how common and how burdensome dependency bugs are. We proceed through a series of small quantitative experiments (mostly) in the ROS ecosystem.

4.1 Pervasiveness of Dependency Bugs

Frequency of Dependency Bugs. We want to understand how frequently dependency bugs appear. We will exploit the material collected in the Iteration 3 (Section 3.1). We want to compute the fraction of dependency bugs within the entire bug population. We will approximate the bug and dependency bugs populations using our simple classifier script from Section 3.1, checking for substrings 'bug' and 'depend' respectively. This classifier is obviously very imprecise, so we first need to estimate its precision.

In the third iteration, we have collected and labeled 50 random issues positively classified by the script. This allows to estimate the probability of a correct classification if the classifier detects a dependency bug (precision). We have removed nine invalid entries out of the 50 during manual labeling ($22 + 19 = 41 < 50$):

$$\Pr(\text{dep-bug} \mid \text{positively-classified}) = \frac{\Pr(\text{dep-bug} \cap \text{positively-classified})}{\Pr(\text{positively-classified})} = \frac{22}{22+19} \approx 54\%$$

The same probability for 4th iteration data is ≈ 49% (95 cases positively classified cases, minus 8 removed as invalid), which shows a reasonable stability. The probability of negative classification (NB. not recall, but the precision of the negative classification) is estimated based on the other half of the dataset in the third iteration:

$$\Pr(\neg\text{dep-bug} \mid \text{negatively-classified}) = \frac{\Pr(\neg\text{dep-bug} \cap \text{negatively-classified})}{\Pr(\text{negatively-classified})} = \frac{38}{38+5} \approx 88\%$$

Here 7 cases have been removed as invalid in the manual labeling.

In conclusion, we can use the simple classifier script to estimate the number of dependency bugs, if we discount positive results by 54%, and include 12% of the negative results.

The space of bug kinds is diverse and large. Any particular kind is bound to appear rather rarely in a random population of bugs. Measuring bug frequencies might be misleading, if we disregard other factors such as workload involved in removing, or potential consequences of not removing them. The point of this exercise is not to show that bugs happen rarely, but that they do happen, and they are not a singleton phenomena, but they are repeatable with regularity.

Table 2 collects the frequency statistics. According to the estimation about 16% of all reported issues in ROS are dependency issues. We contrasted this number with the frequency established using the same method for top 30 GitHub projects, chosen by the number of stars. As we can see this yields almost identical estimated frequency of dependency problems. The most 'dependency positive' project in this population is Kubernetes, exhibiting 343 positively classified issues, with 15% of all issues estimated to be dependency bugs.

Observation 6. *Dependency bugs in ROS (and in top 30 GitHub projects) appear at a noticeable frequency. Estimated 15% of all discussed issues and pull requests deal with dependency problems.*

To provide some context, we have adapted the classifier to find concurrency issues (by matching substrings 'concurren', 'parallel', 'deadlock', 'race', 'lock') and memory management issues (substrings 'leak', 'null dereference', 'buffer', 'overflow'). Since we do not have the manual precision labeling for these bugs, we compare the intensity of discussions on these topics, not the predicted bug frequency (using the classifier directly, without the precision correction). Table 3 summarizes these statistics. Without the precision correction, these data are to be interpreted with much caution. Still, we can clearly see that the mentions of memory specific terms, or concurrency specific terms in bug discussions are similar.

Observation 7. *It appears that dependency bugs are not less visible in software projects than other 'established' kinds of bugs.*

How Widespread Are Dependency Bugs? To investigate this we estimate how many projects and how many developers are affected by dependency problems. First, for each of the 118 projects in ROS Melodic that contain an issue tagged 'bug' we manually check whether at least one of these issues is a dependency bug. Second,

Table 2: Dependency bugs in ROS and GitHub projects

Issue Tracker	Positive	Labeled bug	est. dep-bugs (+54%,+12%)	Freq.
ROS Melodic 118 repositories	152	1354	226.32	16%
Top 30 GitHub projects	1018	19553	2773.92	14%
Kubernetes	343	5029	747.54	15%

Table 3: Dependency vs memory and concurrency issues

Issue Tracker	Positive	Labeled bug	Fraction
ROS Melodic 118 repositories (dependency)	152	1354	11%
ROS Melodic 118 repositories (concurrency)	235	1354	17%
ROS Melodic 118 repositories (memory)	74	1354	5%
Top 30 GitHub projects (dependency)	1018	19553	5%
Top 30 GitHub projects (concurrency)	3115	19553	16%
Top 30 GitHub projects (memory)	2669	19553	14%

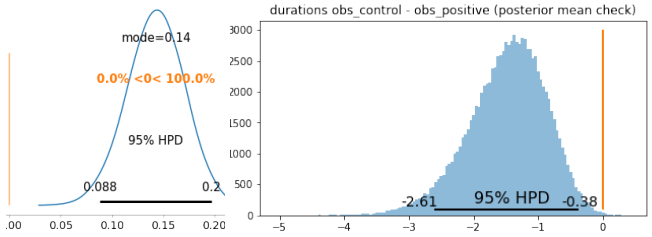


Figure 2: Bayesian analysis of discussion lengths. Left: differences between posterior discussion termination rates for the two groups; Right: posterior difference between length of discussions in the two groups

we check what fraction of developers involved in 'bug' issue discussion in ROS projects were also involved in the dependency bugs discussions as indicated by our simple classifier.

Observation 8. *As many as 53% of packages and 30% of contributors involved in reporting or removing bugs in ROS are affected by dependency problems, so these problems are widespread.*

In an average ROS package (with an issue tagged 'bug'), 30% contributors participate in discussions about dependency bugs (out of contributors discussing bugs at all). This fraction raises to 60% for packages where at least one dependency bug was discussed (so removing zeroes), showing that these bugs tend to attract discussion and time from multiple contributors, in fact, the majority of the package team.

4.2 Cost of Dependency Bugs

With the above two observations, we have to admit that even if the dependency bugs are conceptually simple, they must induce a noticeable cost on the development project, with so many discussions being centered around them, involving so many contributors. We investigate this issue deeper by the intensity of discussions on dependency bugs, their reach.

Many dependency bugs discussions are very short. A developer reports a problem, and another (or the same one) fix it. Significant outliers exist though. A dependency issue report in ROS-Industrial core component[14] has turned into a 22-post long debate with many references from (and thus repercussions in) other threads. The discussion involved two most senior developers in the ROS-Industrial community, and revolved about major architectural choices. Since dependencies are used to enforce and materialize the system and ecosystem architecture, this is bound to happen once in a while.

We measured the lengths of discussions for the manually labeled dependency bugs and compared them to the average lengths of bug discussions in ROS. The average discussion of dependency bugs includes 4 comments, including the first report, with the standard deviation of 4.09; This number is 2.92 ± 3.42 for the entire population of bug discussions in ROS Melodic projects. Figure 2 summarizes the Bayes analysis for the two populations. The left graph shows the posterior probability of differences between parameters of geometric distributions learnt from the treatment group (dependency bug discussions) and the control group (all other bug discussions). In the right, we see the posterior difference between the actual discussions length in the two groups. In both cases, the difference zero lies outside the 0.95 credible interval, so we reject the null hypothesis that two datasets could have been generated by the same distribution.

Observation 9. *Dependency issues in ROS attract more discussions than other issues (on average), and the duration of these discussions is more diverse, which indicates more long outliers.*

Our qualitative inspection of discussions reveals participation of many senior community members. Senior members tend to identify and fix issues quickly. A common picture is a senior contributor reporting a dependency issue, immediately followed by a pull request. However seniors also get involved debugging and fixing dependency issues reported by novices. Many dependency discussions can also be seen on ROS answers,[15] a StackOverflow-like online community for ROS, where novices ask questions and seniors respond. Thus, even though, that dependency issues are a big problem for novice contributors, they do generate a significant cost for the key project members.

Observation 10. *Dependency issues generate a considerable interaction noise for senior community members.*

5 RQ3: FINDING DEPENDENCY BUGS

A lot is to be learnt about bugs by building bug finding tools. The tool builder perspective is different than that of a programmer.

5.1 Method

We settled to prototype three ad hoc linters,[16] for runtime dependency bugs. Runtime bugs are interesting as they may strike in the relatively late life cycle phases, even after the package is released, on users' machine. We looked into historical bugs in the ROBUST database [24] and identified three kinds of runtime dependency errors for inspiration: missing runtime dependencies for shell scripts, missing runtime dependencies for Python imports, and missing runtime dependencies for system configuration and startup files. These three appeared relatively generic, in the sense that any complex system could experience them, albeit with different concrete file formats. The impedance between the deployment manifests and other scripts is likely a common experience.

We wanted to to build simple tools, as project-specific tools need to be developed on a limited budget. After the linters had been built, we ran them on the head of the development branch of all 406 packages in ROS Melodic Morenia, as of 2019/4/3. We explain in detail, what kinds of bugs are found by these linters, how the linters work, how complex they are, and how well they perform, and what this experience teaches us about dependency bugs.

5.2 Prototype Bug Finders

For each linter we report: (i) the category of bugs it finds, (ii) an example bug in ROS Melodic, and (iii) how the linter has been built.

Shell Script Dependency Bugs. ROS packages use shell scripts to set up or to run system commands as part of installation and operation. If a program used in a script is not installed the execution fails, possibly leaving the package in an inconsistent state. This raises

[14] https://github.com/ros-industrial/industrial_core/issues/46

[15] http://answers.ros.org

[16] Source available at: https://github.com/itu-square/ROS-Dependency-Checkers

```
#/bin/bash
source $ROS_ROOT/../rosbash/rosbash
roscd cob_script_server/src
ipython -i cob_console.py
```

(a) An example script from the `cob_script_server` package in ROS

```
<package>
  <name>cob_script_server</name>
  <version>0.5.1</version>
  <description>The cob_script_server package provides a simple
  interface to operate Care-O-bot...
  ...
  <run_depend>rospy</run_depend>
  <run_depend>message_runtime</run_depend>
  <run_depend>actionlib</run_depend>
  <run_depend>actionlib_msgs</run_depend>
  <run_depend>tf</run_depend>
  <run_depend>std_msgs</run_depend>
  <run_depend>cob_srvs</run_depend>
  <run_depend>trajectory_msgs</run_depend>
  <run_depend>geometry_msgs</run_depend>
  <run_depend>control_msgs</run_depend>
  <run_depend>move_base_msgs</run_depend>
  <run_depend>cob_sound</run_depend>
+ <run_depend>ipython</run_depend>
</package>
```

(b) The package manifest for `cob_console`. The missing dependency on ipython is added in the line marked with +

Figure 3: A dependency bug involving a shell script

a missing dependency bug—row 2, 5, or 7 of Table 1, depending on what the broken shell script is used for. The used program must be explicitly listed in the manifest file (`package.xml` in ROS), so that the installation system (rosdep) can install it as a prerequisite.

Figure 3 presents an actual example of such a bug in the package `cob_script_server` from the Care-O-Bot stack. The script (Fig. 3a) invokes the `ipython` interpreter in the very last line, but the manifest (Fig. 3b) has no runtime dependency on `ipython`. The script crashes with a 'command not found' error when executed.

In order to detect such bugs we need to identify the programs invoked in scripts. As bash scripts are executed line-by-line, we assume that each command is placed on its own line (true for majority of scripts). We use a parser[17] to generate ASTs and query them for commands. Then we check this list against `package.xml` and the list of available commands in a minimal ROS installation in an official Docker container. Any missing names are flagged as errors. The implementation of the prototype is 132 lines of TypeScript code.

Python Dependency Bugs. A Python import resembles dynamic linking: it makes a library available or raises a runtime exception. An exception may leave the package or the robot in an inconsistent or unsafe state. Listing all imported modules in the package manifest ensures that they are installed along with the package itself.

Figure 4a shows a program from the package `cob_command_gui` of the Care-o-Bot stack. The program imports modules `pygtk`, `gtk`, and `pynotify`; all missing in the manifest `package.xml` (Fig. 4b). This program may fail with an exception on the first invalid import.

Python imports can be analyzed by generating ASTs, isolating imported modules and checking these against the built-in system modules in a minimal installation and the list in `package.xml`. Our implementation of this idea takes about 10 lines of Python and 50 lines of shell script. It uses an external Python analyzer, flake8.[18]

```
#!/usr/bin/python
# Project name: care-o-bot
# ROS package name: cob_command_gui
# Implementation of ROS node for command_gui.
import roslib
roslib.load_manifest('cob_command_gui')
from cob_msgs.msg import EmergencyStopState
from command_gui_buttons import *
import thread
import pygtk
pygtk.require('2.0')
import gtk
import roslib
import os
import pynotify
import sys
import signal

planning_enabled = False ...
```

(a) A Python program importing pygtk, gtk, pynotify (abbreviated)

```
<package>
  <name>cob_command_gui</name>
  <version>0.5.1</version>
  <description>This package provides a simple GUI
                          for operating Care-O-bot.</description>
  <license>LGPL</license>
  <url type="website">http://www.ros.org/wiki/cob_command_gui</url>
  ...
  <buildtool_depend>catkin</buildtool_depend>
  <run_depend>rospy</run_depend>
  <run_depend>cob_script_server</run_depend>
  <run_depend>cob_relayboard</run_depend>
  <run_depend>python-pygraphviz</run_depend>
+ <run_depend>gtk</run_depend>
+ <run_depend>pygtk</run_depend>
+ <run_depend>pynotify</run_depend> ...
</package>
```

(b) The manifest file for `cob_command_gui` lacking (marked +) the dependencies imported by the Python script above

Figure 4: Runtime dependency missing: imports in Python

Launch File Dependency Bugs. A launch file in ROS is a script in a DSL (XML-based) that defines the configuration of the system, composes the necessary nodes, and starts them up. Launch files are executed by an interpreter, `roslaunch`, which calculates arguments and passes them to the invoked parts of a ROS-based system. Packages use the `find` directive in the launch files to resolve locations of other packages and of their launch files. If a dependency is accessed in a launch file (i.e., via the find `directive`) but not declared in the `package.xml`, it will not be present in the runtime system, and the launching of the robot software will fail.

Figure 5 shows an example of such a bug in a driver for a Universal Robot arm. The launch scripts configures a simulator node (under some conditions), and the physical robot (otherwise, not shown). However the simulator package is not a dependency in the manifest file, so it will not be automatically installed on machines of developers using this package. As soon as they start the driver node with the simulator argument, the system will crash.

[17] https://github.com/vorpaljs/bash-parser

[18] https://gitlab.com/pycqa/flake8

```
<launch>
  ...
  <!-- the "sim" argument controls whether we connect to -->
  <!-- a Simulated or Real robot -->
  <!-- - if sim=false, a robot_ip argument is required -->
  <arg name="sim" default="true" />
  <arg name="robot_ip" unless="$(arg sim)" />
  <!-- load robot specific joint names -->
   <rosparam command="load"
      file="$(find ur5_moveit_config)/config/joint_names.yaml"/>
  <!-- load the robot_description parameter -->
  <include
    file="$(find ur5_moveit_config)/launch/planning_context.launch">
    <arg name="load_robot_description" value="true" />
  </include>
  <!-- run the robot simulator and action interface nodes -->
  <group if="$(arg sim)"><include
    file="$(find industrial_robot_simulator)/launch/....launch"/>
  </group>
  ...
```

(a) The launch file `moveit_planning_execution.launch` configuring the industrial robot simulator node (last lines)

```
<package>
  <name>ur5_moveit_config</name>
  ...
  <run_depend>moveit_ros_move_group</run_depend>
  <run_depend>moveit_simple_controller_manager</run_depend>
+ <run_depend>industrial_robot_simulator</run_depend>
  ...
```

(b) The package manifest missing dependency declaration on the industrial robot simulator

Figure 5: Launch dependency bug in a driver for a robot arm

We build a linter for this class of bugs, following the procedure from the previous two examples. First, we identify the find objectives in launch files using an XML parser (lxml). Then we compare them against the dependencies available in the minimal ROS installation and those listed in the package manifest. The implementation takes 109 lines of Python code.

Observation 11. *Since most dependency bugs are relatively simple, it is feasible to script lightweight syntactic linters at rather low cost, acceptable for any sizable project.*

It appears that we cannot even build a single checker for runtime dependency bugs (see above), as the linters depend heavily on the kinds of artifacts that host the dependencies. Thus each entry in Table 1 may require many linting procedures in practice.

Observation 12. *The linters depend heavily on project-specific file layout and deployment information, thus it is difficult to generalize these tools to apply to a wider set of ecosystems.*

Table 4: Running the linters on 406 packages of ROS Melodic

Linter	Relevant packages (out of 406)	#found bugs
Shell script dependencies	65	5
Python import dependencies	204	0
Launch file dependencies	174	5

5.3 Results

Table 4 summarizes the results of running the three above linters on all packages in the ROS Melodic distribution. The second column lists how many packages were relevant for the linter: how many contained (respectively) any shell scripts, any Python programs, and any launch files. The first and the last linter find five new bugs each. The Python linter finds no bugs (on the day of running the experiment). We assessed that all the found issues are real, but three were deemed very minor. We have reported seven issues to the project issue trackers. Of these seven, four issues have been fixed, one was rejected, and two remain open as of today.

What do we learn from this? Even though we have built the three bug finders based on historical bugs, we have no guarantee that these bugs are frequent. Building a dependency tool may yield good results, but understanding which tool to build is hard. This is especially so as measuring frequency of detailed kinds of historical bugs is very tedious—in fact, more laborious than scripting linter procedures for many of them.

Observation 13. *It may be relatively easy to script project-specific linters for dependency bugs, however it is challenging to decide which linters should be implemented to maximize the benefit.*

6 VALIDITY

Internal Validity. Our experiments are approximate and limited. We classified only 224 out of 1354 bug-tagged issues in ROS Melodic, using keyword matching. We implemented linter prototypes only for some bugs. To mitigate the first issue we used substring search and simple statistical estimation to automatically analyze all 1354 issues. Mitigating the second issue is difficult, and requires building a research agenda on general bug finders for dependency bugs.

Building the taxonomy in Table 1 proceeded in parallel with coding. All but two out of 22 codes have been identified in the first three iterations. The largest fourth iteration (about half of the coded positive cases) only added two codes, and no new groupings. We believe that this is a sign of saturation; all large categories of dependency bugs in ROS have been spot. It is possible that other systems display other categories. The taxonomy might be inexhaustive.

Not all top GitHub projects (Section 4.1) are software projects. To mitigate this, we selected only software projects, filtering out teaching and political material, and various catalogs.

The constructed linters are neither sound or complete. We reported the 10 issues and we received confirmation (with varying levels of severity).

External Validity. This is a case study of dependency bugs in ROS, with limited experiments outside. Most observations are strictly ROS-specific, until proven otherwise. Other subject ecosystems could lead to different conclusions. Still, ROS is a rather large and diverse ecosystem, with heavy use of both static and dynamic programming languages, along with multiple project-specific DSLs, generated and interpreted code, runtime introspection, modeling tools, driver software, AI components, with both user-oriented graphical tools and headless components. It is concurrent and distributed, synchronous, asynchronous, reactive. As such, it reflects modern software systems reasonably well. We believe that similar studies performed on other platforms would lead to similar results.

7 RELATED WORK

Software bugs have been studied extensively [2, 11, 15, 26, 27]. Many studies focus on particular categories of bugs, such as design flaws [4], security holes [21], variability [1], or floating-point bugs [7]. These studies illustrate severity and characteristics of the bug categories, and drive much research in building automated tools or methodologies, such as static or dynamic analyses [5, 8].

However, research on dependency bugs is rare. Kerzazi et al. [13] study failures of automated builds. Our assessment of the situation is consistent with theirs, however we note that only about half of the observed problems manifest at build time; dependency bugs reach further than build. We have previously shown that supporting developers in debugging cross-language links reduces bugs of this kind [18]. Resolving dependencies is similar to name resolution as formalized by Neron and coauthors [17]. Their theory might be a good starting point towards a general solutions to dependency problems. Handling dependencies is a known subject in software configuration management [14, 20]. None of these work provides a thorough study of dependency bugs comparable to the presented one.

Large ecosystems, such as Android and Linux, provide an interesting study ground for bug studies [1, 6]. ROS is another complex software system involving multiple languages and DSLs, hardware interfaces, and non-determinism [19]. Not much has been written on bugs in ROS though. Kate et al. propose a tool for detecting errors in use of physical units in calculations in ROS [12]—a rather different problem; within, not across, programming languages and packages. HAROS aggregates and interfaces several off-the-shelf static checkers for ROS [23]. It extracts an abstract model of a ROS project, which has been used to execute empirical studies [22]. We believe it could support implementation of linters and further empirical studies. None of these reports discusses the dependency bugs though.

Our linters are pattern-based analyzers akin to Pylint[19] and FindBugs [10]. They are designed to be lightweight, scaling to the entire ROS codebase, unlike traditional static analyzers based on abstract execution of program semantics [16].

8 CONCLUSION

We presented a study of dependency bugs in ROS, a rich large open source ecosystem for robotics. Despite relative simplicity, the dependency bugs are surprisingly common phenomenon in committed code (~15% bug reports). They disturb multiple developers, often involving senior community members. We prototyped building simple linters, and observed that they do find bug instances in real code, yet it is hard to predict which linters will be effective. Despite being data-driven, one of our linters found no bugs.

We hope to attract the attention of the research community to dependency bugs. We posit that the redundancy of specification of dependencies, and the impedance between the different specification locations, is common also outside ROS (in pip, gem, Eclipse OSGI, apt, etc.). It is an annoying and time consuming problem; conceptually simple, and, unlike many other correctness issues in software, possibly feasible to eliminate. On the other hand, it is an evil problem: many aspects of it are specific to the software architecture and tools at hand. This makes it hard to produce general tools and general results for finding and fixing these bugs.

[19] https://www.pylint.org, seen Feb 2020

Acknowledgements. We thank ROSIN EC Horizon 2020 grant 732287 for support; Raúl Pardo for the Bayesian analysis; Chris Timperley, Gijs van der Hoorn, Andre Santos, Harshavardhan Deshpande, and Jonathan Hechtbauer for the ROBUST dataset that motivated us.

REFERENCES

[1] I. Abal, C. Brabrand, and A. Wąsowski. 2014. 42 variability bugs in the Linux kernel: a qualitative analysis. In *ASE'14*. ACM.
[2] P. E. Black. 2018. A Software Assurance Reference Dataset: Thousands of Programs With Known Bugs. *Journal of research of the NIST* 123 (2018).
[3] S. Chitta, I. A. Sucan, and S. Cousins. 2012. MoveIt! *IEEE RAM* 19, 1 (2012), 18–19.
[4] M. D'Ambros, A. Bacchelli, and M. Lanza. 2010. On the impact of design flaws on software defects. In *International Conference on Quality Software 2010*. IEEE.
[5] G. Doychev, B. Köpf, L. Mauborgne, and J. Reineke. 2015. Cacheaudit: A tool for the static analysis of cache side channels. *ACM TISSEC* 18, 1 (2015), 4.
[6] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su. 2018. Large-scale analysis of framework-specific exceptions in Android apps. In *ICSE'18*. IEEE.
[7] A. Di Franco, H. Guo, and C. Rubio-González. 2017. A Comprehensive Study of Real-world Numerical Bug Characteristics. In *ASE'17*.
[8] P. Godefroid, M. Y. Levin, and D. Molnar. 2012. SAGE: whitebox fuzzing for security testing. *Commun. ACM* 55, 3 (2012), 40–44.
[9] G. Gousios. 2013. The GHTorrent dataset and tool suite. In *MSR '13*. 4.
[10] D. Hovemeyer and W. Pugh. 2007. Finding more null pointer bugs, but not too many. In *Workshop on Program analysis for software tools and engineering*. ACM.
[11] C. Jones and O. Bonsignour. 2011. *The Economics of Software Quality, Portable Documents*. Addison-Wesley Professional.
[12] S. Kate, J.-P. Ore, X. Zhang, S. G. Elbaum, and Z. Xu. 2018. Phys: Probabilistic physical unit assignment and inconsistency detection. In *ESEC/FSE'18*.
[13] Noureddine Kerzazi, Foutse Khomh, and Bram Adams. 2014. Why Do Automated Builds Break? An Empirical Study. In *ICSME'14*.
[14] Nir Koren, Run Profesorsky, and Ido Itzkovich. 2017. Source code change resolver. US Patent App. 14/932,513.
[15] S. Sahoo Kumar, J. Criswell, and V. Adve. 2010. An empirical study of reported bugs in server software with implications for automated bug diagnosis. In *ICSE*.
[16] A. Miné, L. Mauborgne, X. Rival, J. Feret, P. Cousot, D. Kästner, S. Wilhelm, and C. Ferdinand. 2016. Taking Static Analysis to the Next Level: Proving the Absence of Run-Time Errors and Data Races with Astrée. In *ERTS'16*. Toulouse, France.
[17] P. Neron, A. Tolmach, E. Visser, and G. Wachsmuth. 2015. A theory of name resolution. In *ESOP'15*. Springer.
[18] R.-H. Pfeiffer and A. Wąsowski. 2012. Cross-language support mechanisms significantly aid software development. In *MODELS'12*. Springer.
[19] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. 2009. ROS: An open-source Robot Operating System. In *ICRA workshop on open source software*, Vol. 3.
[20] N. Ratti and P. Kaur. 2018. A Conceptual Framework for Analysing the Source Code Dependencies. In *Advances in Computer and Computational Sciences*.
[21] E. Rescorla. 2005. Is finding security holes a good idea? *IEEE Security & Privacy* 3, 1 (2005), 14–19.
[22] A. Santos, A. Cunha, N. Macedo, R. Arrais, and F. Neves dos Santos. 2017. Mining the usage patterns of ROS primitives. In *IROS'17*.
[23] A. Santos, A. Cunha, N. Macedo, and C. Lourenço. 2016. A framework for quality assessment of ROS repositories. In *IROS'16*.
[24] C. Timperley and A. Wąsowski. 2019. 188 ROS bugs later: Where do we go from here? https://vimeo.com/378916121 ROSCON'19.
[25] B. Vasilescu, Y. Yu, H. Wang, P. T. Devanbu, and V. Filkov. 2015. Quality and productivity outcomes relating to continuous integration in GitHub. In *ESEC/FSE'15*.
[26] H. Zhang and S. Kim. 2010. Monitoring software quality evolution for defects. *IEEE software* 27, 4 (2010), 58–64.
[27] L. Zhao and S. Elbaum. 2003. Quality assurance under the open source development model. *Journal of Systems and Software* 66, 1 (2003), 65–75.