PRABHAB: Change Impact Analysis for Package Management Systems

Joseph Hejderup
Delft University of Technology
The Netherlands
j.i.hejderup@tudelft.nl

***Abstract*—Package management systems such as NPM, MAVEN, or CARGO facilitate large collections of modular FOSS libraries, called packages, that act as reusable building blocks for software systems. While packages have a small code footprint, they have dependencies on other remotely-developed packages, making package repositories highly interconnected. As a side effect, ill-thought or inevitable changes in new releases on existing packages can negatively impact inter-dependent packages in a repository, creating problems such as cascading build breakages. In this position paper, we propose Prabhab, a novel approach to aid package maintainers in estimating the impact of code changes against a package repository using PRÄZI, a fine-grained dependency network. Further, we discuss potential applications that Prabhab can aid in quality assurance of package releases and package repositories. Finally, we present a set of challenges in realizing this approach.**

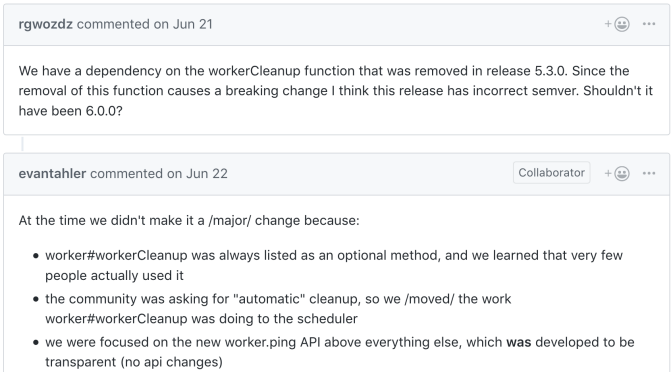


Fig. 1: Breaking change for `taskrabbit/node-resque`, #252.

I. INTRODUCTION

Package managers provide a central mechanism to make the rich diversity of community-developed OSS libraries, hosted in centralized code repositories, available into the workspace of software projects. Central to package managers are packages; these are archives of bundled library functionality with a declared manifest, providing a standardized format to compose and build upon library functionality from remotely-developed libraries (known as a *dependency*). Inspired by the Unix philosophy [1], a package should ideally be small, do one thing well and act as a building block for other packages [2]. By making libraries small, modular and distributable through a packing system has amplified the development towards vivid and large code repositories.

A side-effect of this fast-paced development is the formation of strongly interconnected packages as a result of depending on many small remotely-developed packages [3], [4]. As many software systems and packages within a repository also subscribe for automatic updates of their package dependencies [5]–[7], an update of a dependency can trigger an adverse impact to connected packages within a package repository. Upon publication of a new release, new changes do not undergo external testing nor code reviewing, making it prone to break update compatibility (e.g., breaking API changes) and introducing bugs (e.g., memory leaks). As a result, a mistake will affect dependents but also spread to other connected dependents, creating a domino effect in the package repository. The risk is real; a study suggests that certain packages have the power to impact 30% of a package repository [8].

With a growing number of dependencies in projects [8], localizing and dealing with such changes is both daunting and a time sink for development teams [9] as exemplified in Figure 1. As a consequence, good intentioned package maintainers may risk losing its active user-base due to ill-thought changes. To mitigate this risk, package repositories recommend the use of semantic versioning to label the degree and character of the changeset [10]–[12]. However, misclassification and misinterpretations of changes are common [7]. Another compelling method is the use of regression testing tools such as `NoRegrets` [6], `dont-break` [13] and `cargo-crusador` [14] which are capable of detecting breaking changes in dependent packages. While useful, these tools are limited to detection of breaking changes on directly depending users, making it incomplete by leaving out non-breaking changes such as bugs and how changes impact indirect dependent packages. Thus, impeding the comprehension of understanding the complete impact of a set of code changes in a package repository.

In this paper, we propose a novel method for change impact analysis of package management systems by emulating the effect of package release roll outs using a dependency network to obtain a more complete and precise impact set. Instead of using conventional dependency networks, our technique performs analysis on PRÄZI [15], a scalable versioned call-based dependency network to identify impacted repository paths of inter-package function calls. The resulting impact set constitutes of impacted call paths which are ranked and

classified according to the popularity of the package and also its centrality in the package repository. The impact set enables package maintainer to understand to what extent changes affect a package repository, and whether those entities are important to the package repository.

II. Impact transfer in package management systems

A. Impact transfer of a new package release

To make a package available in a software project, developers need to specify the name and version constraint of the package. A version constraint denotes the anticipated set of compatible package releases within a project. Package managers support forward-compatibility in version constraints, enabling automatic updates to the latest compatible release. As an example, installing a package with CARGO generates a dependency descriptor with forward-compatibility by default, as shown in Listing 1. In Listing 1, `my-project` is compatible with `lodash` ^4.17.11, where the caret (^) symbol instructs the package manager to fetch the latest non-major version up to the next major release (e.g., if x is a version, the compatibility range is $x \geq 4.17.11 \cap x < 5.0.0$). In addition to only updating to the latest non-major version, a tilde ($\sim$) symbol prepended to a version (e.g., $\sim$4.17.11 or 4.17.x) updates to the latest non-minor version (e.g., $x \geq 4.17.11 \cap x < 4.18.0$).

```
{
    "name": "my-project"
    "dependencies": {
     "lodash": "^4.17.11"
    }
}
```

Listing 1: `cargo install lodash`

While forward-compatible version constraints provide a mechanism for staying up-to-date with bug fixes of a package dependency, it is also a source of introducing build failures and new untested code to software projects. In relation with many other dependencies in a project, localization, and troubleshooting of failing dependencies become challenging. Figure 2a depicts the dependency tree with inter-procedural function calls of a `ToDo App`, which uses a package to upload new tasks to three online services, trello, evernote and wünderlist. Building the `ToDo App` at *t1* results in an operational application. However, rebuilding the `ToDo App` at a later point in time *t2*, leads to a build failure as shown in Figure 2b. Here, the `validator` dependency is automatically updated (e.g., see 0.2.x) from version 0.2.3 to 0.2.4 with a semver-incompatible change by renaming the method `validate()` to `isValid()`. As a result, `restler` depending on the `validate()` method from version 0.2.3 will result in a breakage which in turn propagates up to the `ToDo App`. On the other hand, despite the semver-incompatibility, `reqwest` does not call `validate()` and continues to be operational.

Given the current tooling for regression testing of dependent clients [6], the hypothetical library maintainers of `validator` would have limited insights about the impact of the changes made in the release of version 0.2.4 for the following reasons: (1) the impact set leave out transitive clients (In Figure 2, a library maintainer may not be aware that a potentially popular `ToDo App` will break because of breaking changes in a less insignificant package, `restler`), (2) the impact does not treat dependency information as a first-class citizen, in particular, forward-compatibility of clients (In Figure 2, had we excluded `restler` or perhaps used a development version, the library maintainer would assume that no one is affected).

The event of automatically updating a dependency of an existing package from a previous version v to a new version v' can have an adverse impact to connected dependent packages without directly visible relations in a package management system. We consider two classes of code changes to be important for impact transfer in a package management system, *structural* and *implementation* changes of a package's public API interface.

a) Impact of structural changes: Structural changes, commonly known as *breaking changes* arise when an updated version is not backward compatible with a previous version of a dependency. This includes changes such as function removal, modified argument list or return type of an API function, and access visibility. An immediate impact of structural differences between two versions of the same package dependency is a build error, which further propagates to depending packages that will also fail to build.

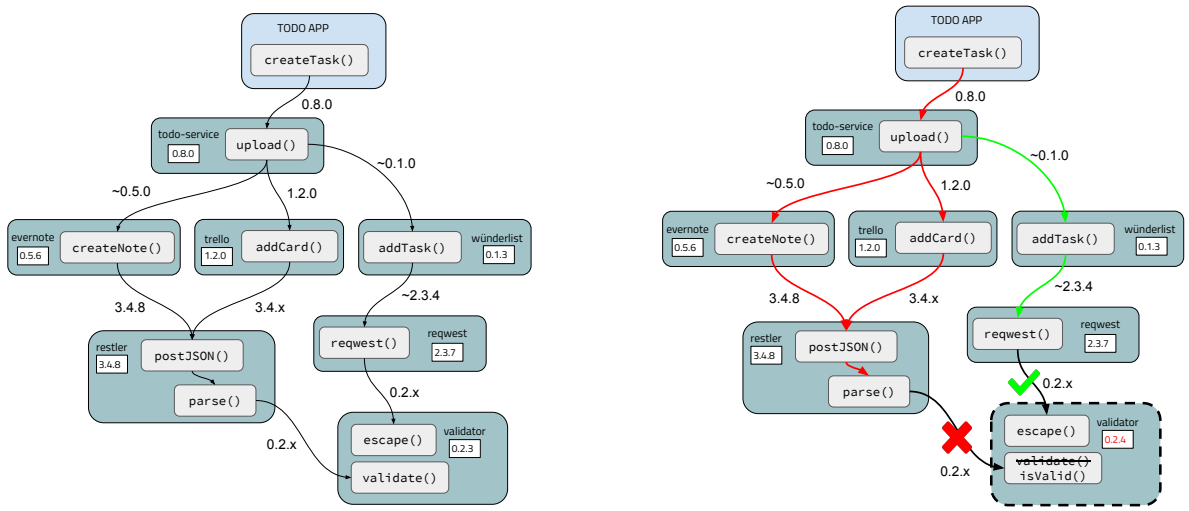
b) Impact of implementation changes: Implementation changes in relation to a public API can be a source for bug introduction which can destabilize a subset of a package repository. While identification of such bugs can be hard to catch, implementation changes that impact a large subset of package repository should receive external testing and code reviewing before being released.

B. PRÄZI: call-based dependency network

Package-based dependency networks can by its nature not provide a fine-grained resolution to perform change impact analysis. To realize PRABHAB, we build on-top of PRÄZI, a call-based dependency network that derives its dependency relationships from function calls in packages instead of conventional metadata [15]. With PRÄZI, we can track control flow changes across a package repository.

III. Impact analysis of package changes

PRABHAB detects propagation of changes in a package repository in a four-phase process as shown in Figure 3. In the first phase (1), PRABHAB emulates a rollout of a proposed release to identify dependents packages that would update to the new release. For each identified dependent (2), PRABHAB generates a diff between its current version and the proposed release to identify a list of changes. Once all changes are identified (3), PRABHAB performs a reachability analysis on



(a) Working tree at *t1*

(b) Failing tree at *t2*

Fig. 2: Dependency tree of a modern todo application

PRÄZI to discover impacted inter-package call chains in the repository. Finally (4), the impact set is ranked according to package importance such as active development, popularity and centrality in a dependency network.

A. Emulating a new release

By emulating a rollout of a new release, we extract the subset of packages in a package repository that would be impacted by an update. To calculate this subset, we construct a versioned dependecy network by resolving all package constraints, and add the emulated release as an available package to resolve to. Then, we do a query on the dependency network to find all connected dependent packages. The resulting set should contain dependents that will resolve to the new version along with the version it currently resolves to.

B. Identifying changed entities

After obtaining the package version each potentially affected dependent resolves to, we perform a diff to identify the source code changes. From the source code changes, we use a diff analyzer to identify whether a change is *structural* or *implementation* and which API endpoints are affected. Then, we use PRÄZI to perform a reachability analysis to retrieve affected inter-package call chains in the package repository. The result of running the reachability analysis for the changeset of each dependent is the impact set.

C. Ranking the change set

To aid package maintainers with additional information about the change set, we extract information such as centrality in the package repository, download popularity and active development of affected packages.

IV. Related Work

Change Impact analysis is a widely studied problem in program analysis research [16], [17]. Propagation of changes in package repositories has become an important research area in light of incidents such as the *left-pad incident*, and recent moves to emulate these problems on package-based networks [2], [8], [18].

a) Regression testing: Regression testing, i.e., a form of testing that aims to confirm that a proposed code change has not adversely affected existing features, is a close area to our work. Closest to our work is `NoRegrets` [6], a tool that detects breaking changes in test suites of dependent NPM packages before releasing an update of the library. On similar lines, the work of Raemaekers *et al.* [7] studies the impact of `semver` incompatibilities of Maven packages, suggesting that one-third of the releases introduces a semver-incompatible change. In contrast to their work, our technique also considers the impact of transitive dependents and aims to calculate the impact against the entire package repository. Moreover, we also consider *forward-compatibility* of dependent clients - a new release may not necessarily result in dependent clients updating and, thus yielding a more accurate representation.

b) Change Impact Analysis: We perform change impact analysis on a call graph representation to compute the impact set. Several techniques [19]–[23] use call graphs as an intermediate representation for change impact analysis. Alternative techniques to call graphs are static and dynamic slicing [24], [25], profiling [26], [27] and execution traces [28]. Due to cost-precision trade-offs, several proposed approaches use a combination of these techniques. One such example is Alimadadi *et al.*'s work on Tochal, that leverages both runtime data and call graphs to more accurately represent changes to dynamic features such as the DOM. While the RustPrazi representation could benefit from dynamic information, the gain is substan-

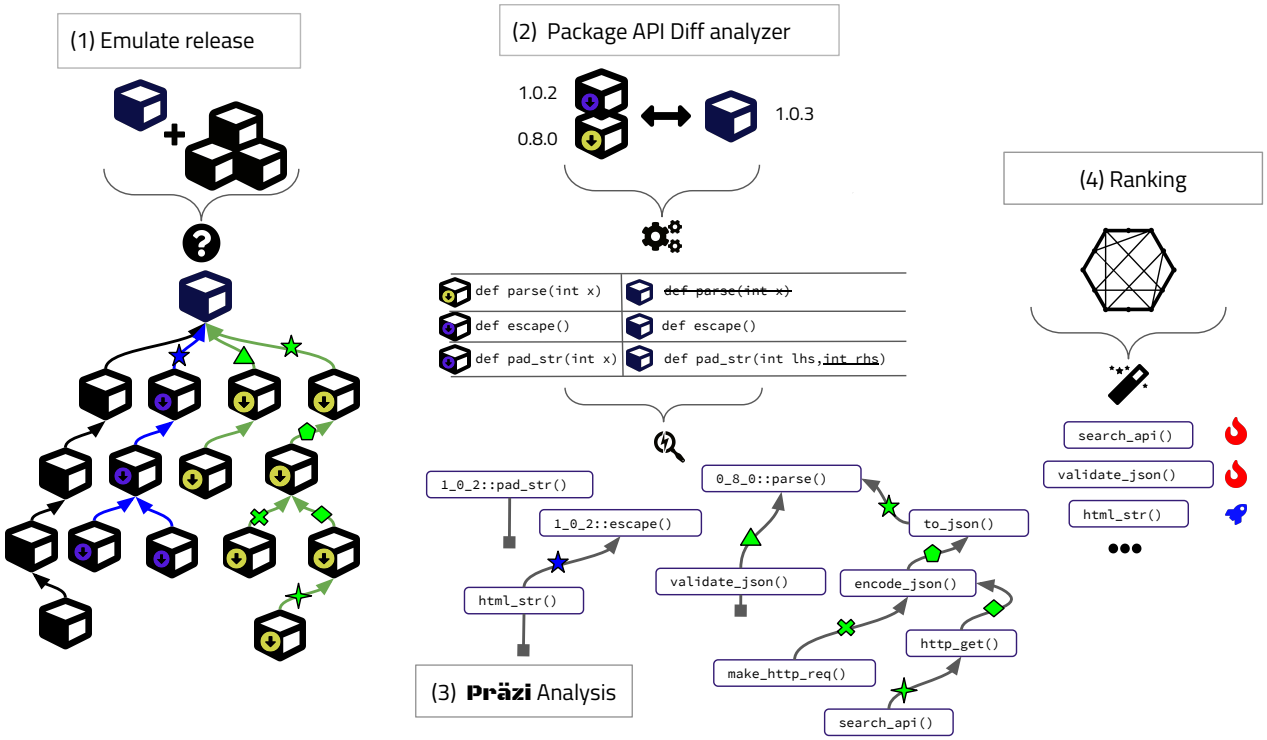


Fig. 3: Our approach to perform change impact analysis of a package repository

tial to the completeness, making dynamic instrumenation an expensive trade-off. For a comprehensive overview of impact analysis techniques and change estimations, we refer the reader to Li *et al*'s [16] survey on code-based change impact analysis techniques.

c) API studies: While change impact analysis prominently proposes techniques with respect to enhancing completeness and precision of results, several papers investigate decisions behind API changes. Sawant *et al* [29] studied the reaction of deprecation of more than 25,000 clients from four libraries. Overwhelmingly, clients do not react to deprecation features. In a follow-up study [30], a missing aspect in the decision making of deprecating features is that there is no well-defined protocol on how to do it. PRABHAB could aid package managers with information about use of deprecated functions to make more informed decisions. Bogart *et al*. [9] conducted interviews with API developers in 3 software ecosystems: Eclipse, npm, and R/CRAN. The three ecosystems have different polices and values with respect to breaking changes. A key insight is that developers monitor changes in dependencies actively, however, current tooling are burdensome due to noise. Moreover, developers are increasingly avoiding the use of forward-compatibility and reducing the number of dependencies to avoid dealing with upstream changes. Package managers can use PRABHAB as part of the publication process; if a set of changes will affect a large part of the repository, the release will be put on hold with need for additional external validation. This way, clients can take advantage of forward-compatibility without being disrupted by breaking changes.

d) Dependency networks: The aftermath of the *left-pad incident* has led to a surge of studies around package repositories. Researchers have constructed dependency networks of package repositories to trace the impact of security problems [8], [18], [31], to study the evolution of language ecosystems [3], [4], [8], or health [2], [32] In the area of security, notably, Kikas et al. [8] have shown there exist packages that can break up to 30% of packages in both NPM and RUBYGEMS. With PRABHAB, our technique can complement such analyses to include propagation of risky code changes in package repositories.

REFERENCES

[1] Malcolm D McIlroy, EN Pinson, and BA Tague. Unix time-sharing system: Foreword. *Bell System Technical Journal*, 57(6):1899–1904, 1978.

[2] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 385–395. ACM, 2017.

[3] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 351–361. IEEE, 2016.

[4] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, Feb 2018.

[5] Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 84–94. IEEE Press, 2017.

[6] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. Type regression testing to detect breaking changes in node. js libraries. In *32nd European Conference on Object-Oriented Programming (ECOOP 2018)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2018.

[7] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

[8] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 102–112. IEEE press, 2017.

[9] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120. ACM, 2016.

[10] Common practices to make your gem users and other developers lives easier. . (Accessed on 21/10/2018).

[11] Alex Crichton. Api evolution. , May 2015. (Accessed on 21/10/2018).

[12] How to use semantic versioning. . (Accessed on 21/10/2018).

[13] Gleb Bahmutov. Do not break dependant modules. , Nov 2014. (Accessed on 09/11/2018).

[14] Test the downstream impact of rust crate changes before publishing. . (Accessed on 09/11/2018).

[15] Joseph Hejderup, Moritz Beller, and Georgios Gousios. Przi: From package-based to precise call-based dependency network analyses. Workingpaper, 2018.

[16] Bixin Li, Xiaobing Sun, Hareton Leung, and Sai Zhang. A survey of code-based change impact analysis techniques. *Software Testing, Verification and Reliability*, 23(8):613–646, 2013.

[17] Steffen Lehnert. A taxonomy for software change impact analysis. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 41–50. ACM, 2011.

[18] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[19] Barbara G Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 46–53. ACM, 2001.

[20] Linda Badri, Mourad Badri, and Daniel St-Yves. Supporting predictive change impact analysis: a control call graph based technique. In *Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific*, pages 9–pp. IEEE, 2005.

[21] Xiaoxia Ren, Fenil Shah, Frank Tip, Barbara G Ryder, and Ophelia Chesley. Chianti: a tool for change impact analysis of java programs. In *ACM Sigplan Notices*, volume 39, pages 432–448. ACM, 2004.

[22] Daniel M German, Ahmed E Hassan, and Gregorio Robles. Change impact graphs: Determining the impact of prior codechanges. *Information and Software Technology*, 51(10):1394–1408, 2009.

[23] Bixin Li, Xiaobing Sun, and Hareton Leung. Combining concept lattice with call graph for impact analysis. *Advances in Engineering Software*, 53:1–13, 2012.

[24] Frank Tip. *A survey of program slicing techniques*. Centrum voor Wiskunde en Informatica, 1994.

[25] Robert S Arnold. *Software change impact analysis*. IEEE Computer Society Press, 1996.

[26] James Law and Gregg Rothermel. Whole program path-based dynamic impact analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 308–318. IEEE Computer Society, 2003.

[27] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 128–137. ACM, 2003.

[28] Alessandro Orso, Taweesup Apiwattanapong, James Law, Gregg Rothermel, and Mary Jean Harrold. An empirical comparison of dynamic impact analysis algorithms. In *Proceedings of the 26th International Conference on Software Engineering*, pages 491–500. IEEE Computer Society, 2004.

[29] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of clients of 4+ 1 popular java apis and the jdk. *Empirical Software Engineering*, 23(4):2158–2197, 2018.

[30] Anand Ashok Sawant, Mauricio Aniche, Arie van Deursen, and Alberto Bacchelli. Understanding developers needs on deprecation as a language feature. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE 2018). forthcoming*, 2018.

[31] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *International Conference on Mining Software Repositories*, 2018.

[32] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE. ACM, 2018.