Dependency Versioning in the Wild

Jens Dietrich
Victoria University of Wellington
Wellington, New Zealand
jens.dietrich@vuw.ac.nz

David J. Pearce
Victoria University of Wellington
Wellington, New Zealand
david.pearce@vuw.ac.nz

Jacob Stringer
Massey University
Palmerston North, New Zealand
jacobstringer@windowslive.com

Amjed Tahir
Massey University
Palmerston North, New Zealand
a.tahir@massey.ac.nz

Kelly Blincoe
University of Auckland
Auckland, New Zealand
k.blincoe@auckland.ac.nz

***Abstract*—Many modern software systems are built on top of existing packages (modules, components, libraries). The increasing number and complexity of dependencies has given rise to automated dependency management where package managers resolve symbolic dependencies against a central repository. When declaring dependencies, developers face various choices, such as whether or not to declare a fixed version or a range of versions. The former results in runtime behaviour that is easier to predict, whilst the latter enables flexibility in resolution that can, for example, prevent different versions of the same package being included and facilitates the automated deployment of bug fixes.**

We study the choices developers make across 17 different package managers, investigating over 70 million dependencies. This is complemented by a survey of 170 developers. We find that many package managers support — and the respective community adapts — flexible versioning practices. This does not always work: developers struggle to find the sweet spot between the predictability of fixed version dependencies, and the agility of flexible ones, and depending on their experience, adjust practices. We see some uptake of semantic versioning in some package managers, supported by tools. However, there is no evidence that projects switch to semantic versioning on a large scale.

The results of this study can guide further research into better practices for automated dependency management, and aid the adaptation of semantic versioning.

I. Introduction

Many modern software systems are built from existing packages (i.e. modules, components, libraries — henceforth, just *packages*). This realises a vision almost as old as software engineering itself: that is, in order to be on par with other parts of a modern economy, re-usable software components must be produced and used at scale [32]. In recent years, many systems have moved to a model where packages are stored in a central repository that is accessed from a client-side *package manager* (e.g. *Maven* for Java, *Cargo* for Rust, *CPAN* for Perl, etc). Such tools access packages as required (e.g. at build time) and, often times, have flexibility over which exact version to use. The use of package managers has considerably simplified the build process (compared with ad-hoc methods used previously) and, furthermore, enabled automatic *package evolution* (e.g. for the controlled propagation of security fixes and other improvements).

One challenge now faced by software developers is deciding, for a given package, on which version to depend. For example, one can depend upon a *fixed version* of a given package (i.e. "use only version 1.2.2") or on a *version range* (i.e. "use version 1.2 or better"). With fixed versions, builds are more deterministic[1], but critical fixes in later versions of the package will not be automatically included [9]. In contrast, version ranges have the disadvantage that builds can now fail if changes between versions are not backwards compatible [10], [37], [41]. On the upside, version ranges allow the package manager to select the "best" version with respect to some metric (e.g. the latest version meeting all constraints). This means new versions which fix bugs, address security vulnerabilities, or improve performance are automatically included whenever dependency resolution takes place. Another benefit of version ranges is that the package manager can handle packages that are included multiple times by intersecting all constraints to find a single match, thereby preventing multiple inclusions of the same package. This problem has become exasperated in recent times as packages have more and more (transitive) dependencies. While some techniques exist to separate those packages at runtime, such as the use of class loaders in OSGi [34] or JavaScript programming patterns to avoid conflicts in the global namespace like jQuery's `noConflict()`, many systems are still prone to runtime version conflicts and the DLL-hell-style bugs [40] resulting from them.

On the other hand, developers also have to understand how incompatible changes between versions arise. Packages are governed by multi-faceted contracts that are often only implicitly stated [2]. This may include *API signatures*, *semantic contracts* (expressed informally or formally through pre- and postconditions), expectations on *performance* and *resource usage*, and non-technical aspects such as *licenses*. For instance, API signatures are often considered as easy to reason about but, even for well-specified, statically typed languages like Java, the situation is surprisingly complex: most developers don't understand the compatibility rules [11], tools

[1]Even when only fixed version dependencies are used, builds are not necessarily guaranteed to be completely deterministic, e.g., a dependency may itself declare further dependencies using ranges.

that try to detect incompatible evolution are incomplete [24], and incompatible evolution that can break client packages is common [10], [37], [20]. Semantic contract violations present an even bigger challenge for detection. For example, updating an API so that some parameter no longer accepts null is not backwards compatible as it is strengthening a precondition. Whilst such a violation could conceivably be detected using some form of non-null static analysis [14], [31], [4], [16], things are less clear for arbitrary contracts (e.g. JML [22], [6], [27]). Of course, testing provides some capability here [5] but, due to its inherent unsoundness, may easily miss violations. Therefore, many issues caused by incompatible evolution of packages may only be detected after deployment and, hence, can be particularly damaging.

In practice, developers have to make trade-offs between those two strategies, balancing the opportunities of optimised systems with the risks of incompatibility errors. *Semantic versioning* has arisen as a popular approach for managing package evolution which uses structured versions of the form "`major.minor.micro`" [36]. The idea is to associate certain compatibility guarantees with changes to parts of this structure. For example, when increasing the `minor` version of a package (e.g. `1.2.3` $\implies$ `1.3.0`), all changes should be backwards compatible with previous versions at the same `major` level. In contrast, incompatible changes are only permitted between versions at the `major` level (e.g. `1.2.0` $\implies$ `2.0.0`). The challenge for developers, however, lies in correctly following this protocol. This is because (as discussed above) incompatible changes are sometimes subtle and hard to spot, even for seasoned developers. Likewise, there is limited tooling available for checking adherence to the protocol.

We are particularly interested in the adaptation of semantic versioning. That is, given the above difficulties in sticking with the semantic versioning protocol, *what do developers do?* They might, for example, simply eschew semantic versioning altogether in favour of fixed versions; or, they might throw caution to the wind, and fully embrace semantic versioning despite the challenges. The aim of this study is to investigate which choices developers make across different package managers. Looking at different package managers gives insight into the ecosystems of specific languages (e.g. Java versus JavaScript) and language paradigms (e.g. static versus dynamic typing).

More specifically, we (1) set out to capture the current practice regarding how developers declare dependencies, and (2) investigate whether and how developers change their approach as projects mature. This is of particular interest as provides some evidence about whether a certain approach is working or not. Finally, as the semantic versioning scheme was proposed to provide an elegant solution for developers struggling to balance the conflicting goals of the different version strategies, we (3) also look into the adaptation of semantic versioning practices. This results in the following three research questions:

RQ1 How do projects declare dependencies?
RQ2 Do projects change their approach as they evolve?
RQ3 How do projects adapt semantic versioning?

II. Methodology

In this section, the methodology used to acquire and analyse data is discussed. We will discuss limitations and threats to validity within the subsections as necessary. In the interest of reproducibility, the scripts used to acquire, process and analyse the data are made available in a public repository (https://bitbucket.org/jensdietrich/lib.io-study/).

A. Dataset Acquisition

We used a data set from `libraries.io`[2] for this study. `libraries.io` is a service that tracks and indexes package releases from 36 package managers [3]. The data set contains dependency data in CSV format that we imported into a PostgreSQL database for further analysis and processing. In particular, the dataset contains a *dependencies* table which has versioned dependencies of packages to other packages. This table has 71,884,555 records with dependency information for packages from the following 17 different package managers: Atom, Cargo, CPAN, CRAN, Dub, Elm, Haxelib, Hex, Homebrew, Maven, NPM, NuGet, Packagist, Pub, Puppet, Pypi and Rubygems.

Atom is a special case as it also allows users to specify dependencies to NPM packages. However, Atom packages are managed in a different repository [4], and we therefore decided to keep those packages in the data set.

B. Classification Categories

Each package manager uses its own syntax in order to declare dependencies, using a combination of package name and *version constraint*. We were mainly interested in the version constraint part that describes which version(s) of a target package can be used, this is usually a pattern that is applied to versions expressed as `<major>.<minor>.<micro>`, sometimes followed by additional tokens, such as build numbers.

In order to conduct a comparative study across different package managers, we developed a taxonomy to represent various ways to declare dependencies. The challenge was to strike a suitable compromise between enabling a fine-grained analysis but still making sure that the concepts used were sufficiently abstract to be applied to different package managers. We developed a set of classifications (defined in Table I) by iteratively reviewing the dependency declarations across all package managers.

C. Classification Rules

For each dependency declaration for all package managers, we mapped the specified version constraints to the classifications in Table I. To enable the analysis of the large dataset, this analysis was automated by developing a set of rules that map regular expressions to the appropriate classifications.

[2]https://zenodo.org/record/1196312/files/Libraries.io-open-data-1.2.0.tar.gz [Accessed: 1 August 2018]
[3]https://libraries.io/about [Accessed: 1 August 2018]
[4]https://atom.io/packages [Accessed: 16 Jan 2019]

TABLE I
DEPENDENCY VERSION CLASSIFICATIONS

classification	description
fixed	A dependency on a fixed version (of another package), such as `1.2.3`.
soft	Dependency on a fixed version, such as `1.2.3`, but the package manager may chose another version in order to resolve dependency constraints, using some notion of closeness or similarity.
var-micro	Uses a wildcard for the micro part of the version string, such as `1.2.*`, indicating that the project may depend on any version of a package with a version number starting with `1.2`. This may include additional bounds, such as `1.2.* , < 1.2.4`.
var-minor	Uses a wildcard for the minor part of the version string, such as `1.*`, indicating that the project may depend on any version of a package with a version number starting with `1`. This may include an additional bound.
any	Indicating that a project may use any version of the package it depends on, the package manager has unconstrained freedom to decide which one.
at-least	Indicating a dependency on any version following a specific version (inclusive or exclusive), such as `[1.2.3,*]`.
at-most	Indicating a dependency on any version up to a specific version (inclusive or exclusive), such as `[*,2.0.0]`.
range	A custom range, such as `[1.2.0,2.0.0)`, indicating that a project depends on any version from `1.2.0` to `2.0.0`. Either range bound can be inclusive or exclusive.
latest	The dependency should always be resolved to the latest version available, possibly with some qualifier (such as latest-stable, excluding beta versions.
not	A dependency is declared that explicitly excludes a certain version, usually this is caused by a known issue in this version.
other	Some custom pattern, for instance, a complex boolean formula combining any of the resolved category.
unresolved	The dependency string contains unresolved variable references. This case occurs because `libraries.io` scans project metadata such as (Maven) POMs without understanding their semantics. In a POM, a dependency might be declared by a reference to a variable declared elsewhere, such as `$project.version`.
unclassified	Default value of none of the mapping rules can be applied.

Syntactic mapping: There are two possible approaches: a syntactic mapping solely based on the syntax used, and a semantic mapping taking into account additional interpretations of version constraints, such as how version information is processed. To illustrate this point, consider a version constraint `>=1.2.0,<1.3.0` (or, using an alternative syntax, `[1.2.0,1.3.0))` . Intuitively, this would be classified as *range*. But semantically this corresponds to *var-micro*, and some package managers offer an alternative shortcut syntax for this, such as `1.2.*`. We decided to use a purely syntactic classification that would classify this constraint as *range* for two reasons. Firstly, it is very hard to accurately implement a semantic mapping, in particular when additional modifiers (such as *beta*, *rc*, ..) are involved that may have special semantics. Secondly, syntax constructs like wild cards are more than just syntactic sugar for the convenience of the programmer, they indicate that this pattern is common and/or supported since developers follow certain rules and conventions that guarantee a certain level of compatibility across versions matching such a pattern. In other words, they are indications of how the community and an individual programmer declaring a dependency have adapted semantic versioning practices. In this sense, `>=1.2.0,<1.3.0` and `1.2.*` are different, and we wanted this difference to be reflected in our analysis.

Defining rules: Rules were developed for each package manager by reviewing its version constraint specification. The rules were defined using a lightweight domain-specific language (DSL) developed for this purpose. Figure 1 shows a sample classification rule for Maven along with a description of the DSL used. If a version constraint matches the regular expression in the rule, the respective classification is used. The full set of classification rules can be found in the repository.

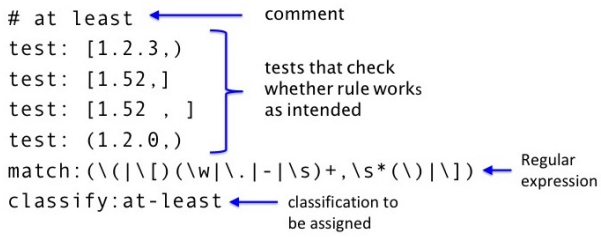

Fig. 1. Sample Classification Rule and description of domain-specific language employed

For a given rule file, rules are applied top to bottom. If none of the mapping rules can be applied, the default classification *unclassified* was used. This usually indicated that the version string is invalid with respect to the package manager specification. Maven is a special case here. The Maven repository and infrastructure is used by different build tools that may use a different syntax to express version constraints. Our classification script (see below) supports the syntax being used by the major package managers (maven, ivy and gradle), but may fail to classify a dependency that uses syntax from a more exotic tool. In this case, *unclassified* is used.

Verifying rules: We verified the accuracy of the classification using the following process. Firstly, the rule DSL has built-in support for writing tests that check whether certain patterns are classified as expected. These tests are checked before the classification is computed, and the process fails if any of the tests fail. Secondly, rule sets were peer-reviewed. For this purpose, a script was developed that created random samples containing version constraints and their associated classifications (obtained using the regular expressions described above). These were manually peer-reviewed and compared against the specification of the respective package manager. The size of the samples was set to obtain a 95% confidence level and a 5-10% confidence interval, additional samples were generated for classifications that were under-represented in the sample to ensure at least 10 records for each classification used for a particular package manager were reviewed. The rules were revised as needed and the peer-review repeated until all version constraints in the sample were correctly classified.

D. Classification Aggregation

While the classification scheme provides a fine-grained view on the various patterns used, it is sometimes useful to consider dependency versioning from a more abstract point of view where we are interested to distinguish between the declaration of fixed versions and variable versions of some kind. Since one of the goals of this study is to investigate the uptake of semantic versioning, we also consider syntax that directly supports semantic versioning practices.

The aggregation of classification categories is defined by the following set of rules, using a simple rule syntax:

```
SEMVER := var-micro | var-minor
FLEXIBLE := range | soft | any | latest | not |
            at-least | at-most
FIXED := fixed
OTHER := other | unresolved | unclassified
```

The semantics of the rules is straight forward: if a dependency is classified using any category in the body (right side) of the rule, then it is classified in the category in the head (left side) of the rule.

E. Version Ordering

In order to answer RQ2, it was necessary to identify the first and the last version of each project in the dataset. Using a naive lexicographical order of version strings is not sufficient to achieve this with a sufficient level of accuracy, for instance, while this would yield `1.2.3 < 1.2.4` as expected, this method would also result in `1.2.10 < 1.2.9`. We therefore opted for an more accurate approach to first sanitise version strings (removing leading "`r`" or "`R`" preceding version strings), then to tokenise leading substrings matching `\d+(\.\d+)*` and comparing versions by comparing those numerical tokens from left to right. This was then implemented in a script that produced a table consisting of project name, first version, and last version for each package manager. Those tables were then sampled and peer-reviewed in order to ensure a sufficient level of accuracy.

We did not consider the semantics of additional strings following the numeric parts of the version, (such as "`-alpha`", "`-beta`", or "`rc1`"), and used the lexicographical order for those suffixes. The reason for this decision was that there is a large number of custom prefixes (including hashes referring to commits), that are often platform– and project-specific. This can lead to cases where our method may not be able to extract the very first or the very last version in the dataset. For instance, we infer `1.2.3-ga < 1.2.3-rc` which is incorrect if one takes the meaning of the respective suffixes (`-ga` – general availability, `-rc` – release candidate) into account. However, we are confident that the first version our approach extracts always precedes the last version.

F. Adaptation of Semantic Versioning

To address RQ3, we needed to identify the stance package managers take towards semantic versioning and the tools used to support semantic versioning. To do this, we canvassed specifications and other documentation for the relevant package managers looking for explicit statements about semantic versioning and its application; likewise, we exercised the tools themselves to check what level of support they provided and, finally, we explored the availability of third-party tools and libraries to support semantic versioning. To do the latter, we conducted two searches for each package manager. First, we searched online using the Google search query: "`PKGMGR (semver OR "semantic versioning")`", where "`PKGMGR`" was replaced with the name of the package manager in question (e.g. Cargo); secondly, we located the associated online package repository and used its search tool with the query "`semver "semantic versioning"`". In both cases, we restricted ourselves to exploring the first twenty results reported [5]. We also considered any tool that was mentioned in the responses to survey question 4 (see Section II-G).

G. Survey

To supplement the analysis of the `libraries.io` data and gain some insights into the intention of programmers using a particular strategy, a survey was created that asked developers about their declaration habits for dependency management. The survey was circulated by email during August 2018 to the authors' industry contacts, and often was further disseminated by the participating developers to others in their companies and beyond. It was also presented to a local professional developer group, where several developers present participated, and posted on public forums such as HackerNews. Since we did not have access to the member lists of some of the groups we used and posted to open forums, we were not able to measure the response rate.

Survey participants: The 170 responses came from a broad range of locations across Europe, North America and Australasia, in addition to about 20 responses from other locations. There was a broad range of experience levels: 0-2 years (9), 2-5 years (27), 5-10 years (43), 10-20 years (66) and 20 or more years (24). We also asked participants which package managers they had used (results in Table II), which shows good coverage of the package managers represented in the `libraries.io` data set. There is only a single system none of the participants is using (Dub). Many respondents reported they use multiple package managers (86 use 3, 48 use 4, and 20 use 5).

TABLE II
PACKAGE MANAGERS USED BY SURVEY PARTICIPANTS.

Package Manager	# Participants	Package Manager	# Participants
NPM	90	Atom	8
Pypi	56	CRAN	7
Homebrew	55	Elm	4
Maven	46	Puppet	4
Rubygems	29	Hex	2
Cargo	22	Haxelib	1
NuGet	22	Pub	1
CPAN	15	Dub	0
Packagist	11	Other	50

Survey design: In addition to the developer metadata discussed above, the survey then asked the following questions:

1) How familiar are you with Semantic Versioning? (On a scale of 1-5)
2) How do you declare dependencies to libraries?
   a) Always using fixed version.
   b) Always using version ranges.
   c) Both depending on the context.
   d) Adopting the styles of others (e.g. copy paste).

[5]The searches were respectively conducted on 17th and 18th Jan, 2019.

3) Has your approach to declaring dependencies changed over time?
4) Do you use any tools to help you version your code?
5) Additional comments about your approach to dependency management.

Free form responses were provided to answer those questions in more detail where the respondent wished to elaborate. These proved valuable in gaining insight into the thought processes that underpin the decisions developers make when managing dependencies.

III. How projects declare dependencies (RQ1)

Table III shows how the version constraints were classified for each package manager using the methodology described in Sections II-B and II-C. The table also contains the number of records in the data set for each package manager in column two, where each record represents a single dependency of a version of a package to some version of another package. It is notable that the number of records varies by a factor of over 10^4 between the package managers with the smallest (Homebrew, 4,886) and the the largest number of records (NPM, 52,886,593).

Considering the data in Table III, the kind of dependency versioning syntax used differs significantly between package managers, and no common pattern emerges. Table IV provides a more abstract view on the data, using the aggregation rules discussed in Section II-D. It appears that there is a preference towards some kind of flexible dependency versioning declaration in all package managers, with significant uptake of a semantic-versioning style syntax in Atom, Cargo, Hex, NPM and Rubygems.

We note however that we only measured the syntax being used, not the intent of the engineer. In particular, there is one package manager where those two aspects may not be aligned – Maven. In Maven, 85.7% of dependencies are declared using the soft version syntax. We think that many developers look up libraries using the maven repository search engine (https://mvnrepository.com/), and copy and paste dependency declaration snippets into the project's `pom.xml` or equivalent build files (for gradle, ivy, etc). These snippets use the soft version syntax, and it is not clear (1) how many developers actually understand that this is not a fixed version and (2) how often Maven resolves this to a different version than what is declared in actual builds using dependency management, mediation and exclusions [6]. We believe that many developers are not aware of the difference and Maven will, in most cases, resolve the reference to the very version declared, indicating that Maven is actually an example of a system where developers take a conservative approach that favours fixed versions.

Elm and Homebrew stand out as both use only one particular versioning strategy. All Elm dependencies are declared using the version range syntax. This is the only syntax supported [7], and the dependency version is generated by the `elm package install` command. This is consistent with the overall approach of Elm to automate versioning and to limit the control developers have. See Section V for more information about how Elm adapts semantic versioning.

All Homebrew packages use the any syntax, granting full flexibility to the package manager to resolve dependencies. This is despite Homebrew offering a syntax for versioned dependencies – a minimum version can be declared as additional dependency information [8].

RQ1 How do projects declare dependencies ?
All package managers investigated use predominantly some form of flexible dependency version syntax, whilst some systems make extensive use of semantic versioning syntax. We note, however, that for Maven, it is unclear whether developers use the flexible versioning syntax intentionally.

TABLE IV
AGGREGATED DEPENDENCY VERSION CLASSIFICATION

	total	FIXED	FLEXIBLE	OTHER	SEMVER
Atom	215,433	17.69%	5.19%	1.33%	**75.78%**
CPAN	2,406,593	0%	**99.99%**	0.01%	0%
CRAN	277,856	0%	**100%**	0%	0%
Cargo	350,862	2.92%	7.89%	0%	**89.19%**
Dub	11,410	6.92%	**59.26%**	8.6%	25.21%
Elm	16,450	0%	**100%**	0%	0%
Haxelib	5,776	39.87%	**60.13%**	0%	0%
Hex	50,227	7.24%	7.37%	3.86%	**81.52%**
Homebrew	4,886	0%	**100%**	0%	0%
Maven	3,592,035	0.03%	**86.85%**	13.05%	0.07%
NPM	52,886,593	16.48%	4.54%	0.68%	**78.3%**
NuGet	3,097,666	6.91%	**93.09%**	0%	0%
Packagist	4,178,062	11.41%	**65.7%**	0.66%	22.23%
Pub	119,810	1.69%	**96.73%**	1.57%	0%
Puppet	57,292	5.79%	**90.03%**	0.21%	3.97%
Pypi	126,536	11.78%	**88.19%**	0.03%	0%
Rubygems	4,487,068	4.59%	**51.12%**	0%	44.29%

IV. Changing Dependency Versioning Practices as Projects Evolve (RQ2)

In order to answer RQ2, we extracted the first and the last version of each project, which were identified using the methodology described in Section II-E. We then compared the dependencies declared in the first and last version. We analysed only the projects that had at least two versions, since this allowed us to investigate evolution of declared dependencies over time. Table V shows some metrics, including the number of projects and the number of projects with only one version for each package manager. The majority of projects have more than one version in the data set, with the exception of the projects using Homebrew.

Table V also shows the average number of versions per project and the respective standard deviations. Those numbers indicate that projects are typically represented by large version ranges, with a significant variation between projects. For instance, there are 33 NPM projects with 1,000 or more versions

[6]https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html [Accessed: 18 December 18]

[7]Although there is no schema that formally defined the syntax of the package manifest `elm-package.json`

[8]https://docs.brew.sh/Formula-Cookbook [Accessed: 18 December 2018]

TABLE III
DEPENDENCY VERSION CLASSIFICATION

	total	fixed	soft	var-micro	var-minor	any	at-least	at-most	range	latest	not	other	unresolved	unclassified
Atom	215,433	17.69%	0%	18.53%	**57.26%**	1.76%	2.32%	0.1%	0.08%	0.93%	0%	1.26%	0%	0.07%
CPAN	2,406,593	0%	0%	0%	0%	**63.14%**	36.84%	0%	0%	0%	0%	0.01%	0%	0%
CRAN	277,856	0%	0%	0%	0%	**80.41%**	19.58%	0.01%	0%	0%	0%	0%	0%	0%
Cargo	350,862	2.92%	0%	**72.86%**	16.32%	6.37%	1.2%	0.02%	0.3%	0%	0%	0%	0%	0%
Dub	11,410	6.92%	0%	23.07%	2.15%	8.23%	**37.09%**	0%	13.94%	0%	0%	8.59%	0%	0.01%
Elm	16,450	0%	0%	0%	0%	0%	0%	0%	**100%**	0%	0%	0%	0%	0%
Haxelib	5,776	39.87%	0%	0%	0%	**60.13%**	0%	0%	0%	0%	0%	0%	0%	0%
Hex	50,227	7.24%	0%	36.81%	**44.72%**	0%	6.99%	0.02%	0.36%	0%	0%	3.86%	0%	0%
Homebrew	4,886	0%	0%	0%	0%	**100%**	0%	0%	0%	0%	0%	0%	0%	0%
Maven	3,592,035	0.03%	**85.7%**	0.05%	0.03%	0%	0.37%	0%	0.77%	0.01%	0%	0.01%	13.04%	0.01%
NPM	52,886,593	16.48%	0%	21.61%	**56.69%**	2.92%	0.72%	0.01%	0.08%	0.8%	0%	0.58%	0%	0.1%
NuGet	3,097,666	6.91%	0%	0%	0%	0.01%	**87.14%**	0.02%	5.91%	0%	0%	0%	0%	0%
Packagist	4,178,062	11.41%	0.02%	14.21%	8.02%	7.9%	3.39%	0.11%	**54.29%**	0%	0%	0.66%	0%	0%
Pub	119,810	1.69%	0%	0%	0%	17.1%	5.92%	0.1%	**73.61%**	0%	0%	1.57%	0%	0%
Puppet	57,292	5.79%	0%	0.86%	3.11%	0%	**56.64%**	0.35%	33.04%	0%	0%	0%	0%	0.21%
Pypi	126,536	11.78%	0%	0%	0%	**49.51%**	33%	0.71%	4.97%	0%	0%	0%	0%	0.03%
Rubygems	4,487,068	4.59%	0%	14.87%	29.42%	0%	**49.25%**	0.23%	1.62%	0%	0.02%	0%	0%	0%

TABLE V
PROJECT EVOLUTION BY PACKAGE MANAGER (PROJ - PROJECT COUNT, ONE - PROJECTS WITH ONLY ONE VERSION, AVG/STDEV - AVERAGE / STANDARD DEVIATION OF NUMBER OF VERSIONS PER PROJECT, AVG1 / STDEV1 - AVERAGE / STANDARD DEVIATION OF NUMBER OF DEPENDENCIES IN FIRST VERSION, AVGL / STDL - AVERAGE / STANDARD DEVIATION OF NUMBER OF DEPENDENCIES IN LAST VERSION

	PROJ	ONE	AVG	STDEV	AVG1	STDEV1	AVGL	STDEVL
Cargo	11,251	3,236	6.13	9.56	3.85	3.54	4.86	4.39
Maven	63,497	16,952	9.96	23.23	5.03	6.3	5.3	7.01
CRAN	11,646	3,223	5.56	8.75	3.54	3.87	6.05	5.98
Pypi	4,083	935	8.76	14.87	2.71	2.85	3.15	3.25
CPAN	28,015	5,055	7.49	15.04	7.24	9.33	10.87	14.34
Elm	1,273	352	4.43	6.36	2.5	1.75	2.54	1.79
Homebrew	1,806	1,784	1.01	0.13	2.77	2.61	2.77	2.61
NPM	547,338	153,412	7.34	22.52	8.75	16.16	9.76	15.78
Atom	3,845	600	11.18	22.11	2.92	3.79	4.08	5.43
Haxelib	470	188	6.06	9.84	1.8	1.26	1.89	1.31
NuGet	76,775	19,860	12.28	48.76	2.77	3.54	2.96	3.77
Dub	550	144	8.69	18.76	1.58	1.18	1.85	1.8
Packagist	104,585	28,340	7.59	16.48	3.37	3.97	4.04	4.6
Rubygems	119,942	35,671	6.41	15.35	4.19	3.41	4.89	3.95
Hex	3,667	1,248	5.4	8.01	2.14	1.56	2.33	1.7
Pub	2,867	688	9.06	17.72	3.08	2.45	3.95	3.34
Puppet	3,703	956	5.61	8.08	2.01	1.83	2.29	2.31

TABLE VI
ADAPTATION OF FLEXIBLE DEPENDENCIES BY PACKAGE MANAGER (PROJ - PROJECT COUNT, SEMV1 / FLEX1 - PROJECTS USING SEMANTIC VERSIONING / FLEXIBLE DEPENDENCY SYNTAX IN FIRST VERSION, SEM+ / SEM- - PROJECTS ADAPTING / DROPPING SEMANTIC DEPENDENCY VERSIONING SYNTAX BETWEEN FIRST AND LAST VERSION, FLEX+ / FLEX- - PROJECTS ADAPTING / DROPPING FLEXIBLE DEPENDENCY SYNTAX BETWEEN FIRST AND LAST VERSION

	projects	SEMV1	SEMV+	SEMV-	FLEX1	FLEX+	FLEX-
Cargo	11,251	6,981	751	30	7,857	111	16
Maven	63,497	324	23	238	43,133	5	5
CRAN	11,646	0	0	0	8,422	0	0
Pypi	4,083	0	0	0	2,908	81	34
CPAN	28,015	0	0	0	22,960	0	0
Elm	1,273	0	0	0	921	0	0
Homebrew	1,806	0	0	0	22	0	0
NPM	547,338	336,963	10,793	5,680	370,102	5,786	5,208
Atom	3,845	2,698	151	51	2,928	90	37
Haxelib	470	0	0	0	219	20	23
NuGet	76,775	0	0	0	54,981	285	624
Dub	550	158	54	28	342	12	7
Packagist	104,585	27,295	6,140	4,860	71,314	2,593	541
Rubygems	119,942	54,001	5,503	3,200	83,207	474	263
Hex	3,667	2,096	80	50	2,213	56	9
Pub	2,867	0	0	0	2,052	24	7
Puppet	3,703	154	28	46	2,660	41	9

in the dataset, and a further 2,584 projects with between 100 and 999 versions. The project with the most versions is *wix-style-react* — it provides common React UI components, with 3,550 versions (ranging from 1.0.0 to 1.1.3547). The large version ranges reflect the trend towards shorter, often highly automated build and release cycles.

Finally, Table V compares the number of declared dependencies the first and the last version for each project, computed using the methodology described in Section II-E. The data indicates that the number of dependencies significantly increases over time for projects in all package managers except Homebrew, where the number stays constant. If we consider external dependencies as a source of complexity of a system, this confirms Lehmann's first and second law of software evolution: projects evolve and become more complex by doing so [28]. Note that there is some additional hidden complexity as we only measure direct, not transitive dependencies.

A. Project Level Analysis

First, we examined the dependency strategies at a high-level by considering the strategies used across all dependencies for each project. The results are summarised in Table VI. We report the number of projects that use flexible / semantic version style dependency versioning in the first version, and add or drop those dependency versioning strategies, shown by its presence or absence in the last version. This is based on the aggregated classification scheme discussed in Section II-D.

As shown in Table VI, projects tend to stick to their way of doing things, and generally resist change with very few projects introducing new dependency versioning strategies or completely removing existing strategies. When projects do change their strategies, they more often move towards using semantic versioning or otherwise flexible dependency declarations, although there are exceptions (notably, *Maven*).

B. Individual Dependency Level Analysis

To complement this coarse, project-level analysis, we also analysed how individual dependencies change over time as we hypothesized that projects will change their versioning practice for some, but not all, of their dependencies.

Again, we found no general trend towards or away from flexible or semantic-versioning style dependency versioning.

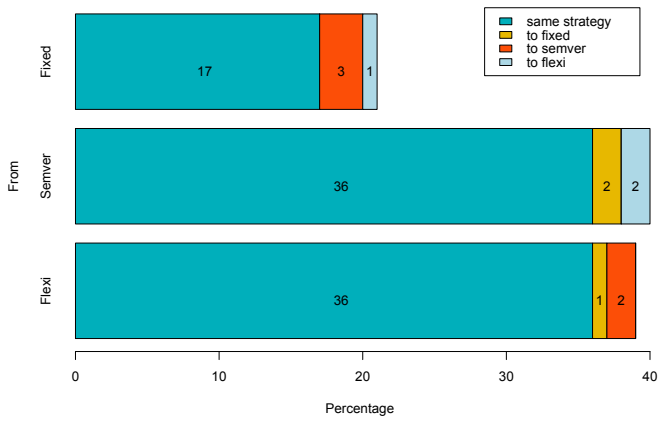

Fig. 2. Changes in Dependency Declarations from First to Last Version

Once a project chooses a dependency strategy for a particular dependency, it is very unlikely that they will change that strategy. Figure 2 shows the strategy changes every time a dependency version declaration is changed. As can be seen, nearly 90% of dependency declaration changes keep the same dependency strategy. This shows that it is very important for projects to consider the implications of these decisions when adding a new dependency.

C. Developer Perspective

Interestingly, the number of survey respondents who report to have changed their dependency declaration approach was rather high (42%). This seems to contradict the results discussed above, we attribute this to the nature of survey participants – we believe that most participants had an over-average level of experience and interest in managing dependencies. Although changes in both directions were reported, changes towards the use of fixed versions were more common [9].

Developers who reported to move away from flexible dependency versioning commonly cited concerns about build stability and the introduction of compatibility-related bugs. The following quotes are illustrative of these concerns: [10]: *"Have been burned too many times by so-called point releases on NPM. "* [10 - 20 years experience, uses NPM, NuGet, Pypi], *"First, realizing that without fixed versions things can break, second change was when I started using cargo which introduced lockfile"* [10 - 20 years experience, uses Cargo, Pypi, very familiar with semantic versioning], *"Taking end-to-end responsibility for software conception, development, and deployment, requires predictable outcomes. If you do not use fixed versions, then rebuilding an artifact to resolve an issue identified during QA testing can cause unrelated changes that can manifest in production."* [10 - 20 years, uses Homebrew, Maven, very familiar with semantic versioning], *"You begin to realize how sloppy upstream people are, and the issues it causes, so you get a bit better about it."* [10 - 20 years, uses Homebrew, NPM, NuGet, Pypi, Rubygems, very familiar with semantic versioning],*"I used to declare version ranges, then I realized that even version ranges cannot capture the full "compatibility range" of a dependency .. "* [10 - 20 years, uses NPM, Nix, very familiar with semantic versioning].

Some experienced developers report differences between package managers, and adapt a horses-for-courses approach *"As much as Python packages claim to implement semantic versioning, they always find a way to break something in a minor release. No issues with Rust though."* [5 - 10 years, uses Cargo, Pypi, very familiar with semantic versioning]. *"Usually due to package maintainers not conforming 100% to semantic versioning resulting in them shipping code that isn't compatible with the previous version. This is most noticeable on npm; .."* [10 - 20 years, uses NPM, Packagist, very familiar with semantic versioning].

This is sometimes also extended to the usage context, with a more conservative approach taken in commercial projects: *"in serious production code, always fixed version... in open source code, more relaxed"* [20+ years, uses Homebrew, Maven, NPM, Rubygems, very familiar with semantic versioning], and organisation imposing rules: *"At work, some packages we are told to keep under a certain range because it is more supported."* [2 - 5 years, uses Homebrew, Maven, NPM, not familiar with semantic versioning].

There were also cases where developers reported their rationale for changing towards flexible dependency versioning practices: *"Because while exact versions give you predictability, they're difficult to keep up to date in manually when you have a lot of dependencies (particularly with pip; pipenv improves on that)."* [10 - 20 years, uses NPM, Pypi, very familiar with semantic versioning], *"Used to be fixed, but .. I wanted to keep up to date and using version ranges helped to do that automatically."* [5 - 10 years, uses NPM, NuGet, Pypi, Others], *"At the start, it felt the easiest to just use a library and keep the fixed version. However, it ended up being quite limiting (especially when there's 'so many cool new features' that I couldn't use). I therefore prefer accepting a certain range of versions to keep the software "fresh" for a longer time."* [5 - 10 years, uses CRAN, NuGet, not familiar with semantic versioning].

[9] In the survey, we have only asked "Has your approach to declaring dependencies changed over time?", and respondents had the option to provide additional text to elaborate, so the above statement is based on these comments

[10] Each quote is followed by some summary data about the respective respondent: years of experience, package managers used, and level of familiarity with semantic versioning

TABLE VII
INDICATIONS OF HOW SEMANTIC VERSIONING IS ADAPTED BY PACKAGE MANAGER (SIG-SIGNIFICANT, SV - SEMVER, SH. - SHALLOW)

	Explicit Statement	Default Tooling	Other Tooling		Supported Styles			Versioning Style Uptake		
			Sh.	Deep	Fix.	Flex.	SV	Fixed	Flex.	SV
Atom	None	Shallow	✓	×	✓	✓	✓	sig.	low	high
CPAN	Weak	None	✓	×	✓	✓	×	low	high	none
CRAN	Weak	None	✓	×	✓	✓	×	none	high	none
Cargo	Strong	Shallow	✓	✓	✓	✓	✓	low	low	high
Dub	Soft	Shallow	✓	×	✓	✓	✓	low	high	sig.
Elm	Strong	Deep	✓	×	×	✓	×	none	high	none
Haxelib	Strong	None	✓	×	✓	✓	×	sig.	high	none
Hex	Strong	Shallow	✓	×	✓	✓	✓	low	low	high
Homebrew	None	None	×	×	×	✓	×	none	high	none
Maven	None	None	✓	✓	✓	✓	✓	low	high	low
NPM	Soft	Shallow	✓✓	✓	✓	✓	✓	sig.	low	high
NuGet	None	Shallow	✓✓	✓	✓	✓	×	low	high	none
Packagist	Soft	Shallow	✓✓	✓	✓	✓	✓	sig.	high	sig.
Pub	Strong	Shallow	✓	×	✓	✓	×	low	high	none
Puppet	Soft	Shallow	×	×	✓	✓	✓	low	high	low
Pypi	Soft	None	✓	×	✓	✓	✓	sig.	high	none
Rubygems	Soft	Shallow	✓	×	✓	✓	✓	low	high	sig.

RQ2 Do projects change their approach as they evolve? The number of dependencies increases as projects develop across all package managers investigated except for Homebrew, where it stays constant. Projects both adapt and drop flexible and semantic version-style dependency version declarations, although the number of projects changing strategy is relatively small. Feedback from experienced developers in the survey suggest that they often move away from flexible versioning as they have experienced compatibility-related errors, and value stable (reproducible) builds.

V. ADAPTATION OF SEMANTIC VERSIONING (RQ3)

The results from Section III support the idea that semantic versioning practices are being applied. For example, consider the results for Cargo in Table III which indicate a large proportion of *var-micro* dependencies. In other words, a lot of dependencies are declared as requiring *any* version of a package with the same `major.minor` number. This suggests developers are assuming backwards compatibility between packages on the `micro` level at least. However, we are interested in whether or not a *stronger* connection with semantic versioning can be established. And, more specifically, how different ecosystems adapt the concept of semantic versioning to their needs.

Table VII provides the results of our analysis looking at how semantic versioning is adapted in practice. As discussed in Section II-F, for each package manager, we determined whether or not explicit statements were made about using semantic versioning and what tooling is available. An explicit statement is an indication that semantic version should be used in some way. We further classified these into: *strong*, *soft* and *weak* statements, where: a strong statement is a *requirement* that semantic versioning be used; a *soft* statement is a *recommendation* that it should be used; finally, a *weak* statement doesn't explicitly mention semantic versioning, but gives informal rules which have a similar intent.

Default tool support reflects the standard tooling that comes with the package manager, and is classified into: *none*, *shallow* and *deep*. Here, shallow support indicates that syntax checking is performed on dependency versions. For example, Cargo checks that versions conform to the format "`major.minor.micro`" and, likewise, checks dependency patterns (e.g. it accepts "`1.2.*`", but rejects "`1.2.*abc`" and "`1.2.**`"). In contrast, deep support indicates some degree of *static analysis* to provide a stronger assurance that semantic versioning is used correctly. For example, in Elm, the command "`elm bump`" attempts to determine the next appropriate version to use by comparing the current source with the previous version.

Other tool support reflects third-party tooling aimed at supporting the use of semantic versioning. Here, *shallow* tools represent the majority of our findings and are only syntactic in nature (where a single tick indicates $1-10$ tools found, and a double tick $1-20$). Indeed, they almost all boil down to two use cases: libraries for parsing and comparing semantic versions; and, tools for "bumping" semantic versions (i.e. where a user asks to increment *major*, *minor* or *micro* version). An example of something which doesn't fit these categories is the NPM semver calculator[11] which, for a given pattern, identifies which versions match. In contrast *deep* tools represent those which attempt some form of static analysis to either generate the next semantic version, or check the given version is correct. Since these are more interesting, we examine them in detail:

- **(Cargo)** `rust-semverver`[12] is a proof-of-concept tool for checking *backwards compatibility*. This tool compares the exposed API surface between two versions, reporting any violations. For example, if a public method is removed in the later version, then this constitutes a breaking change and requires a different major version.
- **(Maven).** Revapi[13], Clirr[14], the Java API Compliance Checker (JAPICC)[15] and the `semantic-versioning`[16] library are all tools, like `rust-semverver`, for checking backwards compatibility.
- **(NPM)** The tool `semantic-release`[17] provides the main example here, and all others found either extend this or are plugins for it. This tool determines the next suitable version using commit messages to determine the extent of changes. For example, if one puts "`BREAKING CHANGE`" in a commit message, then it increments the major version, etc.
- **(NuGet)** The three tools found here, `SemVer.FromAssembly`[18], `SemVerAdvisor`[19] and `SpiseMisu.SemanticVersioning`[20] all

[11] https://semver.npmjs.com/ , all URLs referenced in this Section were accessed on 16 January 19
[12] https://github.com/rust-dev-tools/rust-semverver
[13] https://revapi.org/modules/revapi-maven-plugin
[14] http://clirr.sourceforge.net
[15] https://github.com/lvc/japi-compliance-checker
[16] https://github.com/jeluard/semantic-versioning
[17] https://www.npmjs.com/package/semantic-release
[18] https://www.nuget.org/packages/SemVer.FromAssembly/
[19] https://www.nuget.org/packages/SemVerAdvisor/
[20] https://www.nuget.org/packages/SpiseMisu.SemanticVersioning/

check backwards compatibility based on exposed API surface.

- **(Packagist)** Here the only tool identified was `semcrement` which "*compares structure definitions of your project's code in between commits*"[21].

We must add a precaution regarding the results obtained for identifying third-party tools. Whilst our search approach has yielded a good range of tools and given insight into the relevant communities, there are certainly tools we are aware of which were missed. These include: Japicmp[22], JDK API Diff[23], JDiff[24] and the `SemanticVersionEnforcer`[25]. While those tools are often conceptually interesting, the fact that they did not occur in search results reflects that they are not widely known and haven't seen significant uptake within the respective communities.

Finally, the last three columns in Table VII provide a compact summary of the dependency versioning syntax observed in Section III, in particular Table IV. Uptake is summarised as follows: >50% – strong, 10-50% – significant, 0-10% – low, 0% – none.

RQ3 How do projects adapt semantic versioning? Most package managers encourage packages to follow semantic versioning, though only a few require it. The majority of package managers perform shallow checking of semantic versions, but very few do anything beyond this. Libraries for parsing and comparing semantic versions are widespread, and there is some evidence of more complex third-party tooling being developed to support proper use of semantic versioning.

VI. Survey Results

The developer survey, described in II-G, provided further insight into how versioning strategies are used in practice. The majority of developers surveyed were familiar with semantic versioning, with 73% responding as either familiar or very familiar. Only 9% responded as being not familiar with it at all. Interestingly, developers tended to self-identify as varying their dependency declaration strategy between fixed and range declarations depending on the situation (45%). 32% of developers always used fixed declarations, 11% always used ranges, 6% followed the styles of others (e.g. copy-pasting declarations from Maven repository), and 5% followed other strategies.

Further comments showed that sometimes developers vary their strategy at a library level, for example if some libraries are perceived as better at maintaining backwards compatibility than others, but also that developers change their strategies between types of projects, such as commercial vs. open-source projects. This result contrasts with the analysed dependency declarations in Tables III and IV, which show most package managers having one main declaration style used by convention.

Developers were also asked a question about whether they have changed their approach to declaring dependencies. In this survey, 42% of respondents had changed their method of dependency declarations. However, when analysing the direction of change, no clear shift to or from Semantic Versioning was discernable. This closely mirrors the results discussed in Section IV.

VII. Related Work

A. Evolution of Software Ecosystems

Gonzalez-Barahona et al. [19] studied the growth of the Debian Linux distribution. They found that the average size of packages remained stable, while the overall size of the distribution has been doubling every 2 years and the number of dependencies increased exponentially. This is consistent with some of our findings in Section IV on how the number of dependencies increased.

Several authors investigated the tooling aspect of precisely extracting and representing dependencies, this includes the work of Lungu et. al for Smalltalk [30], German et. al. work on [18] Debian Linux. Men's et al. looked for evidence of Lehmann's laws of software evolution in the Eclipse ecosystem [33], and found that Eclipse bundles adhere to the laws of continuing change and growth.

Espinha et al. studied web APIs [15] and found a general lack of standards and adherence to best practices. This confirms our finding that practices significantly differ between ecosystems.

Bavota et al. have studied dependencies between 147 Java Apache projects, covering 1,964 releases over 14 years [1]. They studied which changes triggered dependency upgrades (bug fixes trigger upgrades, while changes to interfaces make upgrades less likely), and reported that changes between versions have generally only a limited impact on clients. The authors do not precisely model the dependency resolution mechanism used by the actual systems. An interesting finding is that projects are selective when upgrading dependencies, i.e., if they upgrade, they do not perform all available upgrades.

Bogart et al. studied dependency practices in the CRAN, Eclipse and NPM ecosystems [3]. They found that practices differ significantly between ecosystems, all communities invest in tooling to facilitate the maintenance of dependencies, and policies change over time. We include those ecosystems in our work [26], and the results by Bogart et al. confirm our findings that different ecosystems develop unique characteristics.

Decan et al. studied the topology of dependency graphs in seven ecosystems (which were all included in our study): Cargo, CPAN, CRAN, NPM, NuGet, Packagist, and RubyGems [8]. This study is based on the libraries.io dataset, but does not systematically study how dependencies are versioned. Thus, it is complementary to the work presented here.

[21] https://packagist.org/packages/wick-ed/semcrement
[22] http://siom79.github.io/japicmp/
[23] https://github.com/AdoptOpenJDK/jdk-api-diff
[24] http://javadiff.sourceforge.net
[25] https://www.nuget.org/packages/SemanticVersionEnforcer/

[26] Although we do not study the Eclipse ecosystem directly, we note that many Java libraries that are part of the Maven ecosystem are also released as OSGi bundles

Kikas et al. studied dependencies in the JavaScript, Ruby, and Rust ecosystems [25], and found reveal significant differences across those systems. We make a similar observation with respect to versioning practices in our study.

B. Problems Caused by Evolution

Robbes et al. studied the ripple effect caused by APIs changes in the Smalltalk ecosystem [38]. They found that API deprecation can have a significant impact on the ecosystem in terms of broken dependencies that need fixing. Sawant et al. studied the API deprecation mechanism in Java [39] from the point of view of API producers and consumers and found several shortcomings.

Dietrich et al. uses static analysis to study the impact of problems arising from subtle binary incompatibility issues that can break client applications which are simply relinked, rather than being recompiled [10]. They found evidence that breaking changes are common, although this rarely results in issues affecting actual clients. A separate survey by the same authors of 414 developers revealed a possible cause: even experienced developers have only limited understanding of the rules of binary compatibility, and this leads to a large number of compatibility issues reported and discussed by developers online. Xavier et al. obtained similar results, also studying Java [41]. Raemaekers et al. work is conceptually similar, they studied evolution on the Maven ecosystem [37]. One of their research questions was whether the adherence to semantic versioning principles has increased over time. In order to answer this question, they observed breaking changes in non-major releases and observed a moderate decrease from 28.4% in 2006 to 23.7% in 2011, indicating that developers are becoming more aware of semantic versioning.

Dietrich et al. also looked into semantic changes that effect compatibility [12]. By harvesting contracts from code, they identified semantic changes using evolution data from Java programs in the Maven repository. For instance, the strengthening of a precondition when an API evolves is considered as an incompatible change that may break clients.

Linares-Vásquez et al. studied the impact of the use of fault- and change-prone APIs has on the success of Android applications, and found that it negatively impacted the success of apps [29]. Derr et al. also study library dependencies in Android apps, and found large number of outdated versions of libraries being used that could be easily upgraded, in many cases the versions used have known vulnerabilities [9].

Pashchenko et al. studied the impact of vulnerabilities in open-source libraries and, to get industrial relevance, selected libraries used in commercial SAP products [35]. They found that most (but not all) vulnerabilities can be fixed by simply updating dependencies, which highlights the importance of versioning schemes that facilitate safe, automated dependency resolution. Kula et al. conducted a similar study on GitHub projects using Maven dependencies, and found that 81.5% of the systems studied kept outdated dependencies, and developers are often unaware of vulnerabilities in libraries they depend on [26].

C. Tool Support for Dependency Management

There are several studies on how to migrate client code to adapt to changing dependencies, including the work of Cossette and Walker [7] and Dig and Johnson [13] to classify API changes and to automatically recommend and generate refactorings. Henkel and Diwan [21] propose catchup, a tool to record the refactorings used to change APIs, and use those recorings to adapt clients. In all those papers, smaller sets of Java programs were studied.

Jezek et al. reviewed and benchmarked several API compatibility checkers [24], focusing on Java source and binary compatibility issues studied in their previous work [10]. They also suggested a modified Java compiler that produced byte code that avoids some binary incompatibility issues [23]. This automates the refactoring process suggested by other authors [7],[13] for certain types of compatibilities.

Foo et al. report on an API incompatibility checker that uses static analysis [17]. A statically constructed callgraph is used to detect deep changes that can effect compatibility. The tool works for the Maven Central, PyPI, and RubyGems ecosystems, and the authors report that based on the experiments with the tools, 26% of library versions are in violation of semantic versioning. The analysis suffers from the imprecision of the static analysis being used (VTA).

D. Summary

There is a large body of work on software ecosystems and evolution, and any discussion of selected work has to be selective. We notice the following main points:

1) Empirical studies often use Java (usually Maven, older studies often use Eclipse), some of the other systems we are interested in are either under-represented, or ignored altogether. The only other study we are aware of that tries to capture practices across a wide range of package managers is [8].
2) There is plenty of evidence in existing work that both upgrading and not upgrading dependencies can have an adverse effect through compatibility issues and not-addressed faults in dependencies, respectively.
3) There is ongoing research in how to provision better tools to facilitate dependency management.

VIII. Conclusion

We have studied how developers declare dependencies across 17 different package managers, investigating over 70 million dependencies. We find that many package managers support and the respective communities adapt flexible versioning practices. Additional insights gained from the complementary survey indicate that this does not always work, developers struggle to find the sweet spot between the predictability of fixed version dependencies, and the agility of flexible ones, and depending on their experience, adjust practices. We see uptake of semantic versioning in some package managers, supported by tools. However, there is no evidence that projects switch to semantic versioning on a large scale. Interesting topics for future research include more detailed analysis of what the

technological and social barriers to the wider adaptation of semantic versioning are, and how particular communities deal with this.

References

[1] G. Bavota, G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[2] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins. Making components contract aware. *Computer*, 32(7):38–45, 1999.

[3] C. Bogart, C. Kästner, J. Herbsleb, and F. 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.

[4] P. Chalin and P. R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In *Proc. ECOOP*, pages 227–247, 2007.

[5] K. Claessen and J. Hughes. QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs. In *Proceedings of the International Conference on Functional Programming (ICFP)*, pages 268–279. ACM, 2000.

[6] D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Proc. CASSIS*, pages 108–128, 2005.

[7] B. E. Cossette and R. J. Walker. Seeking the ground truth: a retroactive study on the evolution and migration of software libraries. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 55. ACM, 2012.

[8] A. Decan, T. Mens, and P. Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, pages 1–36, 2018.

[9] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes. Keep me updated: An empirical study of third-party library updatability on android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 2187–2200. ACM, 2017.

[10] J. Dietrich, K. Jezek, and P. Brada. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 64–73. IEEE, 2014.

[11] J. Dietrich, K. Jezek, and P. Brada. What java developers know about compatibility, and why this matters. *Empirical Software Engineering*, 21(3):1371–1396, 2016.

[12] J. Dietrich, D. J. Pearce, K. Jezek, and P. Brada. Contracts in the wild: A study of java programs (artifact). In *DARTS-Dagstuhl Artifacts Series*, volume 3. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[13] D. Dig and R. Johnson. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice*, 18(2):83–107, 2006.

[14] T. Ekman and G. Hedin. Pluggable checking and inferencing of non-null types for Java. *JOT*, 6(9):455–475, 2007.

[15] T. Espinha, A. Zaidman, and H.-G. Gross. Web api growing pains: Stories from client developers and their code. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week-IEEE Conference on*, pages 84–93. IEEE, 2014.

[16] M. Fähndrich and K. R. M. Leino. Declaring and checking non-null types in an object-oriented language. In *Proc. OOPSLA*, pages 302–312. ACM Press, 2003.

[17] D. Foo, H. Chua, J. Yeo, M. Y. Ang, and A. Sharma. Efficient static checking of library updates. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 791–796. ACM, 2018.

[18] D. M. German, J. M. Gonzalez-Barahona, and G. Robles. A model to understand the building and running inter-dependencies of software. In *Reverse Engineering, 2007. WCRE 2007. 14th Working Conference on*, pages 140–149. IEEE, 2007.

[19] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: a case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.

[20] D. Haney. Npm & left-pad: Have we forgotten how to program ?, 2016. https://www.davidhaney.io/npm-left-pad-have-we-forgotten-how-to-program/.

[21] J. Henkel and A. Diwan. Catchup! capturing and replaying refactorings to support api evolution. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 274–283. IEEE, 2005.

[22] B. Jacobs and E. Poll. A logic for the Java modeling language JML. In *Proc. FASE*, pages 284–299. 2001.

[23] K. Jezek and J. Dietrich. Magic with dynamo–flexible cross-component linking for java with invokedynamic. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 56. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2016.

[24] K. Jezek and J. Dietrich. Api evolution and compatibility: A data corpus and tool evaluation. *Journal of Object Technology*, 16(4):2, 2017.

[25] R. Kikas, G. Gousios, M. Dumas, and D. 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.

[26] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue. Do developers update their library dependencies? *Empirical Software Engineering*, 23(1):384–417, 2018.

[27] G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *SCP*, 55(1-3):185–208, Mar. 2005.

[28] M. M. Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980.

[29] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk. Api change and fault proneness: a threat to the success of android apps. In *Proceedings of the 2013 9th joint meeting on foundations of software engineering*, pages 477–487. ACM, 2013.

[30] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*, pages 309–312. ACM, 2010.

[31] C. Male, D. Pearce, A. Potanin, and C. Dymnikov. Java bytecode verification for @NonNull types. In *Proc. CC*, pages 229–244, 2008.

[32] M. D. McIlroy, J. Buxton, P. Naur, and B. Randell. Mass-produced software components. In *Proceedings of the 1st International Conference on Software Engineering*, pages 88–98, 1968.

[33] T. Mens, J. Fernández-Ramil, and S. Degrandsart. The evolution of eclipse. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 386–395. IEEE, 2008.

[34] OSGi Alliance. Osgi core release 7 specification, 2018. https://www.osgi.org/release-7-1/.

[35] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 2018.

[36] T. Preston-Werner. Semantic versioning 2.0.0, 2018. https://semver.org/.

[37] S. Raemaekers, A. van Deursen, and J. Visser. Semantic versioning and impact of breaking changes in the maven repository. *Journal of Systems and Software*, 129:140–158, 2017.

[38] R. Robbes, M. Lungu, and D. Röthlisberger. How do developers react to api deprecation?: the case of a smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[39] A. A. Sawant, M. Aniche, A. van Deursen, and A. Bacchelli. Understanding developers' needs on deprecation as a language feature. In *Proceedings of the 40th ACM/IEEE International Conference on Software Engineering (ICSE 2018)*, 2018.

[40] C. Szyperski. Greetings from dll hell. *Software Development*, 7(10), 1999.

[41] L. Xavier, A. Brito, A. Hora, and M. T. Valente. Historical and impact analysis of api breaking changes: A large-scale study. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 138–147. IEEE, 2017.