NuFix: Escape From NuGet Dependency Maze

Zhenming Li
Software College, Northeastern University
Shenyang, China
lzmneu@163.com

Ying Wang*
Software College, Northeastern University
Shenyang, China
wangying@swc.neu.edu.cn

Zeqi Lin*
Microsoft Research Asia
Beijing, China
Zeqi.Lin@microsoft.com

Shing-Chi Cheung
The Hong Kong University of Science and Technology, and Guangzhou HKUST Fok Ying Tung Research Institute
Hong Kong, China
scc@cse.ust.hk

Jian-Guang Lou
Microsoft Research Asia
Beijing, China
jlou@microsoft.com

ABSTRACT

Developers usually suffer from **d**ependency **m**aze (DM) issues, i.e., package dependency constraints are violated when a project's platform or dependencies are changed. This problem is especially serious in .NET ecosystem due to its fragmented platforms (e.g., .NET Framework, .NET Core, and .NET Standard). Fixing DM issues is challenging due to the complexity of dependency constraints: multiple DM issues often occur in one project; solving one DM issue usually causes another DM issue cropping up; the exponential search space of possible dependency combinations is also a barrier.

In this paper, we aim to help .NET developers tackle the DM issues. First, we empirically studied a set of real DM issues, learning their common fixing strategies and developers' preferences in adopting these strategies. Based on these findings, we propose NuFix, an automated technique to repair DM issues. NuFix formulates the repair task as a binary integer linear optimization problem to effectively derive an optimal fix in line with the learnt developers' preferences. The experiment results and expert validation show that NuFix can generate high-quality fixes for all the DM issues with 262 popular .NET projects. Encouragingly, 20 projects (including affected projects such as `Dropbox`) have approved and merged our generated fixes, and shown great interests in our technique.

CCS CONCEPTS

• **Software and its engineering** → *Software libraries and repositories.*

KEYWORDS

.NET, NuGet, dependencies, empirical study

ACM Reference Format:
Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. NuFix: Escape From NuGet Dependency Maze. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510118

1 INTRODUCTION

.NET is an open-source development platform for building various types of projects (e.g., mobile apps, microservices) implemented using C#, F# and VB [76]. Microsoft provides a tool, NuGet, for building projects and sharing packages on multiple .NET platforms at a central repository *nuget.org* [16]. As of August 2021, *nuget.org* maintains over three million package versions.

With the rapid evolution of the .NET ecosystem, managing dependencies in .NET projects becomes a critical challenge. This is mainly due to the fragmentation of .NET ecosystem, which makes dependency constraints between packages especially complicated: (1) There are nine actively used and updated platform variations (e.g., .NET Framework, .NET Core), involving 220 platform versions [2]. Most .NET packages are specified to some target platform versions while incompatible with other ones. (2) At different target platform versions, a package typically requires different transitive dependencies. As a package evolves, new versions may require different dependencies or support different target platforms.

Due to fragmentation of .NET ecosystem, a project's dependency graph can be greatly changed when developers upgrade/migrate its .NET platform or depended packages. These changes can easily introduce build errors. Based on NuGet's build rules, the package installations of a project will be blocked if there are conflicting constraints (e.g., constraints on package versions and target platforms) imposed by the project or its depended packages. In this paper, we refer to such build errors as ***d**ependency **m**aze* (DM) issues.

Figure 1 shows a real DM issue and illustrates the complication in fixing it. In Figure 1(a), a .NET project received a build error when upgrading its direct dependency `EasyNetQ.DI` to version 6.0.1. This is because the upgrade introduced a new dependency

*Ying Wang and Zeqi Lin are corresponding authors.

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510118

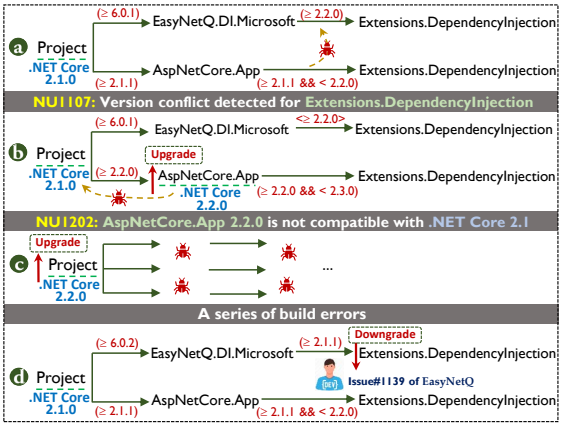

Figure 1: An illustrative example of issue#1139 [5] in `EasyNetQ`

of `DependencyInjection` version (≥ 2.2.0). The new dependency, however, conflicted with the dependency of `DependencyInjection` version (≥ 2.1.1 && < 2.2.0) introduced by `AspNetCore.App`. As a result, the upgrade failed with an error code of NU1107.

To resolve the conflicts, developers first tried to upgrade `AspNetCore.App` to version 2.2.0, which depends on a newer `DependencyInjection` version (≥ 2.2.0 && < 2.3.0) as shown in Figure 1(b). However, a new DM issue arose because `AspNetCore.App` 2.2.0 targets .NET Core 2.2.0 platform, which is incompatible with the project's target platform .NET Core 2.1.0. Therefore, they had to upgrade project's target platform to .NET Core 2.2.0 to resolve such a conflict as shown in Figure 1(c). However, the platform upgrade induced many changes to the project's dependencies. A series of new build errors arose. It is difficult to figure out a solution to resolve all these DM issues, the developers finally rolled back all the changes and filed an issue report #1139 [5] to `EasyNetQ.DI`'s developers, seeking for assistance to downgrade its dependency of `DependencyInjection` to (≥ 2.1.1)) as shown in Figure 1(d). The example illustrates the complexity of DM issues. It also shows that DM issues can manifest themselves in multiple types with respect to different root causes (see § 2.2).

Fixing a DM issue can be challenging. Multiple DM issues often occur in a project at the same time. For those projects with large dependency graphs, fixing such issues is a time-consuming and error-prone process that exercises a series of changes in dependency constraints in response to newly induced DM issues.

Among our collected 169 DM issue reports from 500 representative .NET projects, 42.6% of them involves more than five simultaneous build errors (error count varied from 1 to 150), and 24.9% of them involves more than two error types (see § 3.1). Out of the 169 issues, 124 have documented fixes. On average, each fix requires a combination of 8.8 dependency changes and each change involves 27.5 possible choices in package version constraints, which involves a large solution search space. More importantly, when deciding a fixing solution, developers often consider their desired properties of configuration (e.g., inducing fewer risky build warnings), which are echoed by the discussions in many issues such as #5696 of `aspnetcore` [14]. However, it is difficult for developers to imagine all possible dependency graphs affected by dependency changes, to figure out an optimal solution that satisfies their preferences.

In this paper, we aim to develop a technique to help .NET developers tackle DM issues. We can formulate a DM issue scenario (including installed and available package versions and user upgrade requirements), build rules and developers' preferences as a linear optimization problem to find feasible DM issue fixing solutions. To achieve this, three technical challenges should be carefully studied: (1) how to formally define the configuration preferences in line with .NET developers' practices; (2) how to rigorously assign weights for the preferences encoded in a linear programming model; (3) how to find an acceptable fixing solutions that can induce build warnings (based on NuGet's build rules) if the fully safe solutions do not exist. The existing work [26] adopted a linear programming model to resolve the package upgradeability problem for GNU/Linux-based distributions. However, it cannot be directly adapted to repair DM issues since the three challenges are still unresolved in this approach (see baseline comparisons in § 5.2).

To address the above challenges, we propose an effective technique, NuFix, to repair DM issues. NuFix is built on top of our comprehensive study of developers' preferences in fixing DM issues. Since deriving a developer preferred solution typically requires rich domain knowledge of dependency configurations, we first empirically studied a collection of real DM issue reports, to learn the characteristics of fixing strategies. By encoding the empirical findings into a linear optimization model, NuFix seeks for the global optimal fixing solution. Furthermore, we proposed a variant optimization model based on NuGet's build rules, to seek an optimal sub-solution when a fully safe solution does not exist. Finally, we collected a set of real documented fixes of DM issues by .NET developers as a training dataset to learn the role played by each preference in determining a solution.

Our evaluation shows that NuFix can generate high-quality fixes for the DM issues with 262 popular .NET projects. We invited ten experienced .NET experts working at Microsoft to evaluate the quality our generated fixes. Their feedback indicates that the generated fixes meet the developers' desired properties for the build management. In particular, we further reported our generated fixes to the corresponding project developers for validation. Encouragingly, 20 project developers (including affected projects such as `Dropbox` [19]) have replaced their real fixes with the ones derived by NuFix and shown great interests in our technique.

To summarize, this paper makes the following contributions:

- ***Originality***: To the best of our knowledge, we conducted the first empirical study to learn fixing strategies of DM issues and developers' preferences when deciding fixes.
- ***Technique***: We developed the NuFix tool to efficiently derive fixes for DM issues, satisfying developers' preferences.
- ***Reproduction package***: We provided a reproduction package on NuFix's website (***http://www.nufix-dependency-maze.com/***) for future research, which includes: (1) a dataset of 169 DM issues studied in our empirical study; (2) a benchmark containing 392 documented fixes for real DM issues; (3) available NuFix tool.

2 BACKGROUND

2.1 Fragmentation of .NET ecosystem

Two types of fragmentations exist in the .NET ecosystem.

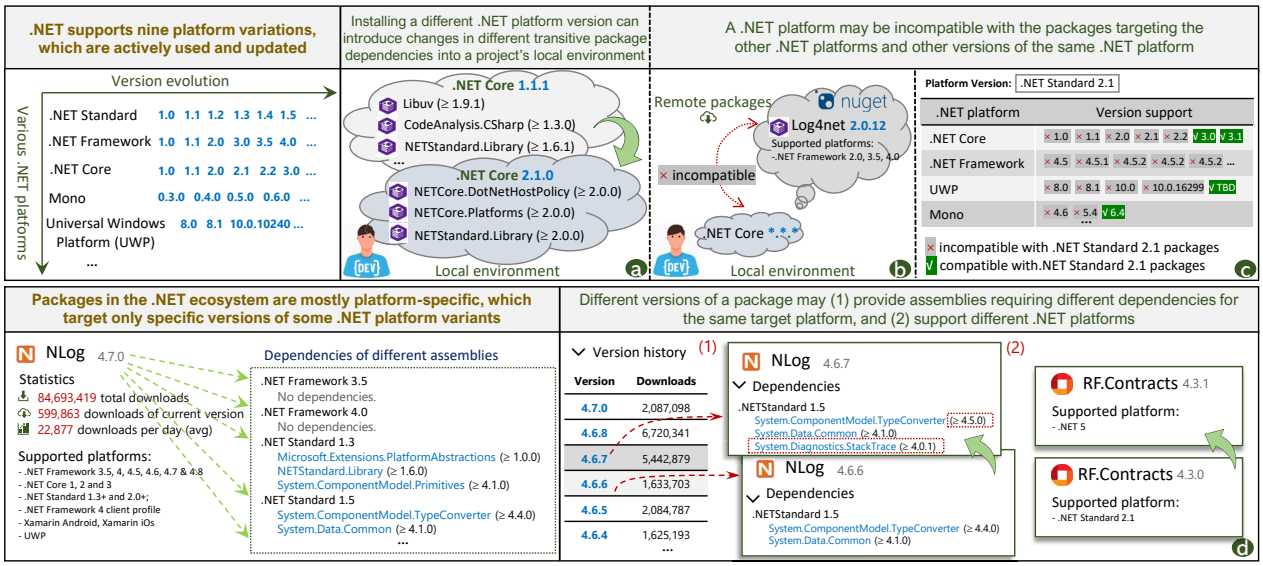

Figure 2: The severity of fragmentation in the .NET ecosystem

- ***Platform-level fragmentation.*** .NET supports nine platform variations, which are actively updated. For instance, five .NET platforms, including .NET Framework, .NET Core, .NET Standard, Mono, and Universal Windows Platform (UWP), were introduced to support Windows OS. By August 2021, the five platforms have released 60 versions [2]. Two evolution characteristics across these versions lead to the platform-level fragmentation problem: (1) *Installing a different .NET platform version can introduce changes in transitive package dependencies with specific version constraints to a project's local environment.* Figure 2(a) gives an example of the change in transitive dependencies introduced by .NET core platform to the local environment from 38 packages to 3 packages after upgrading the platform from version 1.1.1 to 2.1.0.
(2) *A .NET platform may be incompatible with the packages that target other .NET platforms or other versions of the same .NET platform.* For instance, packages targeting one of the four platforms including .NET Framework, .NET Core, Mono, and UWP, are incompatible with the other three platforms. As shown in Figure 2(b), package `Log4net` 2.0.12 is incompatible with .NET Core, since it only supports the .NET Framework ≤ 4.0. To broaden the .NET package ecosystem, since 2016, Microsoft introduced .NET Standard platform for package developers, to provide a unified class library that are available for different .NET variations. Hence the packages targeting .NET Standard can be installed on multiple platforms. Nevertheless, the uniformity of ecosystem has not been realized, since each .NET Standard version only supports limited versions of .NET platforms. Figure 2(c) gives an illustrative example. Based on .NET Standard's documentation [1], a package targeting .NET Standard 2.1 can be installed on .NET Core 3.0 and 3.1, Mono 6.4, and UWP TBD, while it is incompatible with other platforms.
- ***Package-level fragmentation.*** Packages in the .NET ecosystem are mostly platform-specific, which target only specific versions of some .NET platform variants. Each package version provides an isolated *assembly* for each supported platform version. When building a package version, NuGet selectively installs an assembly compatible with the target .NET platform version. The set of dependencies required by each isolated assembly varies. Consider package `Nlog` 4.7.0 described in Figure 2(d). Its assemblies that target .NET Framework 3.5 and 4.0 require no dependencies whereas its assemblies that target .NET Standard 1.3 and 1.5 require 3 and 6 different dependencies, respectively. Moreover, different versions of a package may (1) provide assemblies requiring different dependencies for the same target platform, and (2) support different .NET platforms. For instance, `Nlog` 4.6.6 and `Nlog` 4.6.7 require different dependencies for the same target platform .NET Standard 1.5. In another example, the upgrade of `RF.Contracts` from 4.3.0 to 4.3.1 migrate its target platform from .NET Standard 2.1 to .NET 5.0. These package characteristics induce many variations across various .NET platforms.

2.2 NuGet Dependency Maze

We briefly describe the DM issues numbered by NuGet:

- NU1605 (*Downgrading dependency due to the direct dependency priority rule*): Suppose that different version constraints of the same package are introduced into a project as a direct dependency and transitive dependencies, respectively. Based on NuGet's *direct dependency priority* rule, it installs such a package based on the direct dependency's version constraint. NuGet will report a NU1605 error, if the resolved package version is lower than its transitively introduced version constraint(s).
- NU1107 (*Unable to resolve dependency constraints between transitive packages*): If different version constraints of the same package are introduced into a project as transitive dependencies, NuGet will report a NU1107 error when it cannot resolve a version satisfying all the constraints of conflicting dependencies.
- NU1202 (*Package does not contain any assemblies compatible with the client project's target platform*): If a package does not

contain any assemblies compatible with the client project's target platform, NuGet will report a NU1202 error when installing it.

- NU1203 (*Package does not support the OS where the client project runs*): .NET packages can use configuration attribute *Runtime Identifier* to designate on which OS they can be installed. NuGet will report a NU1203 error, if the *Runtime Identifier* of a required assembly is inconsistent with the OS where client project runs.
- NU1108 (*Circular dependency is not allowed*): NuGet will report a NU1108 error, if a circular dependency (e.g., $\text{PKG}_a \rightarrow \text{PKG}_b \rightarrow \text{PKG}_a$) is introduced into the project.

3 EMPIRICAL STUDY

We empirically study the characteristics of DM issue fixing solutions, with the aim to answer two research questions:

- **RQ1 (Fixing Strategies):** *What are common practices for fixing DM issues? What are challenges of figuring out a fix?*
- **RQ2 (Configuration Preferences):** *What are developers' preferences when determining fixes for DM issues?*

To answer RQ1-2, we collected top popular 3,000 .NET projects on GitHub (by *Star* count) and randomly selected 500 subjects (denoted as $subjectSet_1$) to mine DM issue reports from their issue trackers. We focused on the build failures due to conflicting constraints imposed by build rules. To identify the fixing strategies of DM issues, we analyzed their issue descriptions, developers' discussions and fixes. Note that the rest 2,500 collected .NET projects not used in RQ1-2 (denoted as $subjectSet_2$), were adopted to evaluate our DM issue repair technique later (see § 5).

3.1 Data Collection

Step1: Collecting .NET projects. We collected top popular 3,000 .NET projects from GitHub based on their *Star* counts, and from which we randomly selected 500 subjects as $subjectSet_1$. Figure 3 shows the 500 projects' demographics. They are: (1) non-trivial in size (on average having 63.1 KLOC); (2) well-maintained (on average having 239 code commits); (3) popular (63% of them have over 300 stars); (4) complicated in dependencies (74.1% of them have more than 15 direct dependencies); (5) diverse (concerning projects targeting nine types of .NET platforms).

Step2: Collecting DM issues. We collected DM issues from the 500 projects in $subjectSet_1$. To locate DM issues, we searched the projects for the issue reports filed between May 2018 and May 2021 (i.e., in the past three years) using keywords "version conflicts", "incompatible target platform", and build error numbers (e.g., "NU1107"). Those issue reports that do not have a clear error number were found by the first two keywords. Keywords "version conflicts", "incompatible target platform" and installation error numbers returned 1,631, 922 and 1,012 issue reports, respectively. We removed duplicated reports from the search results and then retained those that explain the issues as well as their root causes.

Eventually, we collected 169 DM issue reports, covering 116 .NET projects. Among them, 124 issues have fixes and 45 issues have clear fixing plans agreed by developers. There are 72 (42.6%) issues that can cause over five build errors. Note that these reports may not contain the new build errors introduced by the fixing process. Figure 4 shows statistics of the 167 issue reports, where N_{IR} denotes the number of issue reports containing each type of DM issues. For

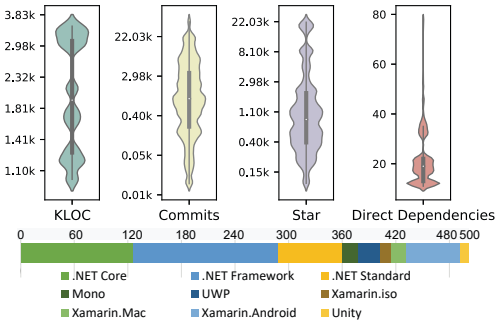

Figure 3: Demographics of the 500 subjects (log scale)

	NU1605	NU1107	NU1202	NU1203	NU1108
N_{IR}	62	89	62	12	6
Max	150	63	16	8	1
Avg	3.58	1.97	1.56	0.18	0.04

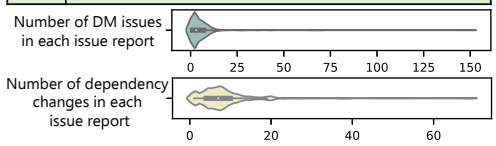

Figure 4: Statistics of the 169 DM issue reports (log scale)

each issue type, *Max* and *Avg* denote the maximum and average issue count in an issue report, respectively. Each report on average describes 7.1 issues, where 42 (24.9%) reports describe more than two issue types. For the 124 issue reports with fixes, the number of involved dependency changes varies from 1 to 69 (i.e., 8.8 ± 8.4).

Step3: Data analysis. To answer RQ1-2, we performed an in-depth analysis on: (1) fixes of the 124 issues, (2) planned solutions of the remaining 45 issues, and (3) comments in the issue reports. We followed an open coding procedure [33], a widely-used approach for qualitative research, to categorize the fixing strategies and configuration preferences discussed in issue reports. Initially, two authors of this paper, who have over three years .NET development experience, independently analyzed the fixes and developers' discussions of 85 issue reports. After the first round of analysis and labeling, the two authors gathered to compare and discuss their results, in order to adjust the taxonomy, with the help of a third author to resolve conflicts. In this manner, we constructed the pilot taxonomy. Next, the first two authors continued to label the fixes or discussions of the remaining 84 issue reports and iteratively refined the results as well as the labeling strategy. The conflicts of labeling were again discussed during meetings and resolved by the third author. We adjusted the pilot taxonomy and obtained the final result.

3.2 RQ1: Fixing Strategies

Via analyzing our collected data, we observed three fixing strategies.

Strategy 1: *Upgrading/downgrading the direct dependencies that directly or transitively introduce the problematic package versions* (116/169). This strategy was adopted by 116 (68.6%) of issue reports to resolve five types of DM issues. For example, in issue#81 [7], project `Installer` directly references package `NETCore.App` with constraint (≥ 2.1.0) and transitively depends on `NETCore.App` again with (≥ 2.1.2) via package `ReverseProxy`. NuGet resolves

`NETCore.App`'s version as 2.1.0, based on the direct dependency's constraint (≥ 2.1.0) (i.e., *direct dependency priority* rule). It then reports a NU1605 error, since `ReverseProxy` has to reference a lower version of `NETCore.App` than it expects. Developers faced the choice of two possible solutions: (1) upgrading `NETCore.App` to 2.1.2; (2) downgrading `ReverseProxy` to a version that was compatible with `NETCore.App` 2.1.0. They adopted solution (2), as their project was incompatible with `NETCore.App` 2.1.2.

While Strategy 1 can fix the original DM issues, it may introduce multiple types of new DM issues. Each direct dependency update can affect the rest of a project's dependency graph, introducing multiple types of new DM issues (e.g., issue#1139 [5] of `EasyNetQ` as discussed in § 1). As a result, developers may need to change more dependencies to resolve the new DM issues.

> 💡 ***Finding 1:*** Upgrading/downgrading the direct dependencies that introduce the problematic package versions is the most common strategy to fix all types of DM issues. The fixes can induce new DM issues.

Strategy 2: *Adding direct dependencies with appropriate version constraints to override the problematic package versions* (61/169). 36.1% of issue reports adopted this strategy to solve NU1107 issues. For example, in issue#886 [8], two conflicting constraints (≥ 10.0.1) and (= 9.0.1) on `Newtonsoft.Json` were transitively introduced by direct dependencies `Tokens.Jwt` 5.2.1 and `Sdk.Functions` 1.0.8, respectively. With the aid of NuGet's *direct dependency priority* rule, developers explicitly declared `Newtonsoft .Json` 10.0.1 as a direct dependency to override its other required versions. After the overriding, these packages can be successfully installed. While the original NU1107 issue was resolved, NuGet generated a new NU1608 warning that the installed version (10.0.1) of `Newtonsoft.Json` exceeded the constraint (= 9.0.1) specified by `Sdk.Functions` 1.0.8. At the end, the developers chose to ignore the warning and enable the `<NoWarn>` property for NU1608 in the configuration file.

We observed that all the 61 issue reports solved NU1107 issues using *Strategy 2* at the cost of inducing NU1608 warnings. As discussed in issue#18 of `ServiceFabric` [6], since its developers could not figure out how to fix their NU1107 issues using *Strategy 1*, they temporarily tolerated the NU1608 warnings induced by *Strategy 2*.

> 💡 ***Finding 2:*** Adding direct dependencies is a common strategy to fix NU1107 issues. However, it often causes NU1608 warnings.

Strategy 3: *Upgrading/migrating the client project's target platform* (34/169). 34 (20.1%) of the issue reports solved NU1202 issues by upgrading/migrating the client project's target platform. For example, in issue#11 [4] of `Unmocakable`, a project got an error when installing `Unmocakable` 3.0.89. The above package only supports .NET Standard 2.1, which is not compatible with the project's target platform .NET Core 2.2.0. By analyzing the interactive compatibility matrix [1] of .NET Standard, developers upgraded their target platform to .NET Core 3.0.0 (compatible with .NET Standard 2.1) in order to work with the desired `Unmocakable` version.

Since packages contain multiple assemblies with different transitive dependencies that support different target platform versions, *Strategy 3* may greatly affect client project's dependency graph. We observed that 8 out of 10 issue reports adopting *Strategy 3* to solve NU1202 issues, introduced a series of new DM issues (e.g., issue#1039 of `refit` [3]). In such cases, developers continued to fix the new issues using a combination of *Strategies 1*, *2* and *3*.

> 💡 ***Finding 3:*** Upgrading/migrating target platform is sometimes adopted to fix DM issues. However, the fixes can have large impact to dependencies and tend to induce new DM issues.

Finally, in 30 out of 169 issue reports, developers cannot figure out fixing solutions. They have to coordinate with the developers of problematic packages to adjust their configurations. We do not consider this solution as a recommended strategy, since it requires developer coordination and is not subject to full automation.

3.3 RQ2: Configuration Preferences

For RQ2, we found clues for developers' configuration preferences in 54 issue reports and summarized as follows:

Avoid inducing incompatibility issues. In 16 of the 54 issue reports (29.6%), to avoid inducing incompatibility issues at runtime, developers tend to adopt a combination of fixing strategies with two preferences below.

- ***Preference 1:*** *Fewer upgrades/downgrades of dependencies' major versions (6/16).* Based on the *semantic versioning strategy* [23], developers are suggested to increase their projects' major version number, if they make incompatible API changes. When exploring DM issue solutions, developers tried to make fewer upgrades/downgrades that change direct dependencies' major versions to avoid incompatibility issues. The above viewpoint was discussed in six issue reports, e.g., issue#972 of `EasyNetQ` [15].
- ***Preference 2:*** *Fewer changes in direct dependencies' versions (10/16).* Developers favored fixing solutions with fewer changes in the version constraints of direct dependencies because this can usually lower the chances that incompatibilities arise after resolving DM issues. Moreover, upgrading is preferable to downgrading if either of them can resolve DM issues. Developers expressed such a preference in ten issue reports, e.g., issue#153 of `AndroidX` [10].

Avoid affecting project maintainability. We observed the following three preferences used in 38 of the 54 issue reports (70.4%) for the sake of project maintainability.

- ***Preference 3:*** *Fewer installed packages (15/38).* Introducing too many packages into a project will complicate the topological structure of its dependency graph. Hence, developers prefer to install the package versions with fewer transitive dependencies, to prevent potential DM issues. Such preference was echoed by discussions in seven reports, e.g., issue#545 of `standard` [13].
- ***Preference 4:*** *Fewer installed packages targeting inconsistent .NET platform with client project (7/38).* Many .NET Standard packages can be directly or transitively introduced into the .NET Core/.NET Framework/UWP/Mono projects. While the projects targeting .NET Standard 2.0 and above can install specific versions of .NET Framework packages. Since each .NET Standard version supports limited versions of other .NET platforms, introducing more packages targeting inconsistent platform with client project would be more likely to cause DM issues when the projects upgrade/migrate their target platforms in future. To ease maintenance, developers try to ensure that more packages introduced in their projects support the same .NET platform as the local ones. Such preference was revealed in seven issue reports, e.g., issue#14393 of `aspnetcore` [9].

- ***Preference 5:*** *Fewer risky build warnings (16/38).* In 16 issue reports, developers expressed that they did not want to adopt the solutions inducing the build warnings below into their projects:
  (1) ⚠ NU1608 (*Package version is outside of dependency constraint*): When different version constraints of a package are introduced as a direct dependency and transitive dependencies, respectively, NuGet resolves such package version based on the direct one. A NU1608 warning will be reported if the resolved version of a package is outside of its other dependency constraints. We observed that developers rolled back fixing *Strategy 2* when exploring solutions for NU1107 issues in five issue reports (e.g., issue #5696 of `aspnetcore` [14]), because they could not tolerate the risky NU1608 warnings induced by such solutions.
  (2) ⚠ NU1103 (*No stable package versions are found in the version constraint*): If there are no stable versions within a package's specified version constraint but only pre-release versions (e.g., *3.0.0-beta-00032*), NuGet will reports a NU1103 warning. Since the pre-release package versions were not reliable, developers did not adopted the fixing solutions introducing such NU1103 warnings in six issue reports (e.g., issue#36550 of `corefx` [11]).
  (3) ⚠ NU1701 (*Package does not specify its target platforms*): If an introduced package does not specify its target platforms, NuGet will reports a NU1701 warning as its assemblies may not be 100% compatible with the client project. To avoid potential incompatibility issues, developers gave up the fixing solutions that caused new NU1701 warnings and adopted other ones in five issue reports (e.g., issue#425 of `GoogleCloudPlatform` [12]).

4 METHODOLOGY

From our empirical study, we understand the pervasiveness and seriousness of the problem, and learn the atomic fixing strategies and developers' preferences from real DM issue fixes. Motivated by our empirical findings (§ 3), the solution design is expected to meet four properties: (1) modeling ability to represent various dependency constraints (§ 2.2), problematic projects with multiple DM issues (§ 3.1), and potential fixes composed of multiple atomic fixing strategies (§ 3.2); (2) learning ability to better adapt developers' preferences by distilling characteristics of real DM issue fixes (§ 3.3); (3) online inference efficiency to derive the fixes within users' acceptable waiting time.

Based on the above analysis, we propose NuFix, a novel technique to derive fixes for DM issues. The key idea is to leverage binary integer linear programming, upon which we can systematically reformulate this task and encode our empirical findings (i.e., fixing strategies and developers' preferences) into linear constraints and optimization functions. Binary integer linear programming is a well-studied technique that meets our efficiency requirements. Moreover, to tune hyperparameters rigorously by learning from real documented fixes, we incorporate Bayesian optimization into our model to optimize the hyperparameters in a non-gradient manner.

4.1 Overall Architecture

Figure 5 gives the overall architecture of NuFix. We list the inputs of NuFix as follows:

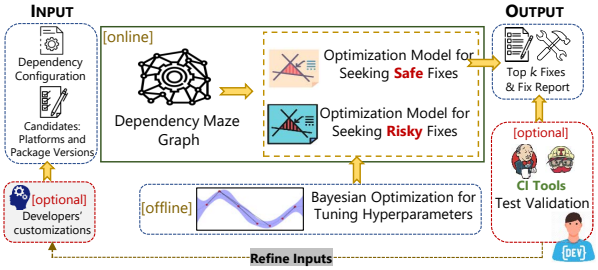

Figure 5: Architecture of NuFix

- *Problematic dependency configuration file (i.e., ∗.csproj).* It configures the client project's direct dependencies and the specified version constraints on them.
- *Candidate target .NET platforms.* It is an ordered list of acceptable platforms, since one of our fixing strategies is to upgrade/migrate client project's target platform (*Strategy 3*).
- *Candidate range of direct dependencies' versions (developers' requirements)* (optional). NuFix allows developers to customize their package upgrade requirements, or the search scope of package versions to avoid inducing undesired program behaviors. For example, developers may want to specify the version of `Lucene.Net` to be limited within $[3.0.3, 4.0.0)$, if they are already aware of the incompatible versions ($\geq$ 4.0.0) through experience. One alternative design is to only consider the package versions without introducing API breaking changes, but this is too strict: it is a common practice that developers may be willing to resolve the DM issues at an acceptable cost of adapting the API changes.

NuFix mainly consists of 5 modules:

- ***Generating Dependency Maze Graph*** (§ 4.2). Given a target platform and the dependency configuration with customized package versions, this module generates a dependency maze graph consisting of all possible dependencies (both direct and transitive) and the constraints among them.
- ***Seeking Safe fixes*** (§ 4.3). Given a DM graph, this module tries to seek the fixes that does not introduce risky build warnings (refer to as *safe fixes*). These fixes are based on *Strategy 1* (upgrading/downgrading direct dependencies), the most commonly adopted strategy in our empirical study (*Finding 1*). To address the combinatorial explosive search space problem, we first propose to formulate the repair task as a binary integer linear programming model, and then we can obtain the fixes using mathematical optimization solvers (in NuFix we use `python-mip` [22]).
- ***Seeking Risky fixes*** (§ 4.4). Sometimes there is no safe fix. Therefore, we propose a variant optimization model to seek fixes that may introduce some risky build warnings (NU1608, NU1103 and NU1701). This model allows not only *Strategy 1* but also *Strategy 2* (adding direct dependencies to override the problematic package versions).
- ***Tuning Hyperparameters*** (§ 4.5). The aforementioned two optimization models involve some hyperparameters. In this module, we tune these hyperparameters based on Bayesian optimization. Note that this is an offline step.
- ***Generating Fix Reports***. For each DM graph, this module reports the top k fixes. To help developers understand the derived

fixes, NuFix also reports their properties (e.g., *a list of build warnings induced by this fix*). The developers can review these fix reports to choose their preferred one.

NuFix also provides a test validation feature that triggers client project's bundled test cases. This helps developers aware of program behavior changes induced by the fix. This is an optional step, as it is time-consuming to re-compile the project and trigger its tests.

4.2 Dependency Maze Graph

Consider a directed graph $G = (V, E)$ with V the set of vertices and $E = \{(i, j)|i, j \in V\}$ the set of edges:

- Each vertex $v \in V$ denotes a package version (e.g., `NLog` 4.6.7). We use two attributes, v.PKG and v.VER to denote the package and the version, respectively. Each edge $e = (i, j) \in E$ indicates that i directly depends on j.PKG, and j.VER satisfies the version constraint specified by i. We exclude the package versions that may lead to NU1202/1203/1108 errors or NU1103/1701 warnings.
- We use a special vertex $v^* \in V$ to represent the client project. For each vertex v satisfying $(v^*, v) \in E$, it means that: this project depends on v.PKG, and v.VER is a candidate version of v.PKG for this project. For each $v \in V$, there is always at least one path from v^* to v. Moreover, to handle the transitive dependencies of platform (Figure 2(a)), the specified platform will also be added as a successor vertex of v^*.

Given G, our main task is to find a set $\hat{V} \subseteq V$, which represents a dependency (both direct and transitive) configuration of the client project that can resolve DM issues. For each $\hat{V}$, NuFix automatically generates the corresponding fix.

4.3 Seeking Safe Fixes

We formulate the repairing task as a binary integer linear programming model as follows.

Variables. For each $v \in V$, we introduce a binary variable $v.\delta$ (i.e., $v.\delta \in \{0, 1\}$), representing whether v should be a dependency (either direct or transitive) in the fix:

$$\hat{V} = \{v|v \in V \wedge V.\delta = 1\} \tag{1}$$

Constraints. First, $v^*.\delta = 1$. Second, each project will not depend on more than one version of the same package (according to NuGet's build rules):

$$\sum_{\substack{i \in V \\ i.\text{PKG}=p}} i.\delta \leq 1, \qquad \forall p \in \{v.\text{PKG}|v \in V\} \tag{2}$$

Third, for each vertex in $\hat{V}$ (i.e., the variable of this vertex takes the value 1), we divide its successors into several groups by the PKG attribute. For each group, there should be one and only one vertex that is in $\hat{V}$. These constraints guarantee that: for each package in the final solution, its dependency requirements will always be satisfied. Formally, we have:

$$\sum_{\substack{i \in V, \\ (v,i) \in E, \\ i.\text{PKG}=p}} i.\delta \geq v.\delta, \quad \forall v \in V, p \in \{v'.\text{PKG}|(v, v') \in E\} \tag{3}$$

Last, transitive dependencies should conform to the *"lowest applicable version"* rule in NuGet dependency resolution:

$$\sum_{\substack{j \in V \\ (j,i) \in E}} j.\delta * I_{\text{LB}}(j, i) \geq i.\delta, \quad \forall i \in \{v|(v^*, v) \notin E\} \setminus v^* \tag{4}$$

where I_{LB} is an indicator function of whether or not i.VER is a lowest application version of i.PKG for j. This constraint guarantees that dependency configuration $\hat{V}$ is entirely based on *Strategy 1* (upgrading/downgrading the direct dependencies).

Objective Function. We design the objective function based on the identified preferences discussed in §3.3.

To encourage *Preference 1* (developers prefer fewer dependency upgrades/downgrades crossing their major versions), we incorporate it into our objective function as a factor:

$$\max z_1 = \sum_{v \in V} v.\delta * I_{\text{MV}}(v) \tag{5}$$

where I_{MV} is an indicator function of whether or not v.PKG is a direct dependency and v.VER shares the same major version as the original version.

To encourage *Preference 2* (developers prefer fewer changes on direct dependencies' versions), we have:

$$\max z_2 = -\sum_{v \in V} v.\delta * I_{\text{U}}(v), \quad \max z_3 = -\sum_{v \in V} v.\delta * I_{\text{D}}(v) \tag{6}$$

where $I_{\text{U}}/I_{\text{D}}$ is an indicator function of whether or not v.PKG is a direct dependency and v.VER is greater/lower than its original version in the problematic configuration file. We use indicator functions I_{U} and I_{D} to distinguish between upgrades and downgrades.

To encourage *Preference 3* (developers prefer to introduce fewer packages in project's dependency graph), we have:

$$\max z_4 = -\sum_{v \in V} v.\delta \tag{7}$$

To encourage *Preference 4* (fewer installed packages targeting inconsistent .NET platform with client project), we have:

$$\max z_5 = -\sum_{v \in V} v.\delta *_C (v) \tag{8}$$

where I_C is an indicator function of whether or not v targets inconsistent .NET platform with client project.

In this section, we ignore *Preference 5* (introducing fewer warnings), because this model will never introduce warning.

We define our overall objective function as:

$$\max z = \sum_{k=1}^{5} \alpha_k * z_k \tag{9}$$

where $(\alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5) \in [0, 1](\sum_{i=1}^{5} \alpha_i = 1)$ are hyperparameters. §4.5 details the tuning hyperparameter procedure.

4.4 Seeking Risky Fixes

Sometimes it is infeasible to find safe fixes. Therefore, in this section we propose a variant model that can seek risky fixes (i.e., fixes that may introduce build warnings). As introduced in § 3.3, we need to consider three types of build warnings: (1) NU1608 (*package version is outside of dependency constraint*); (2) NU1103 (*no stable package versions are found in the version constraint*); (3) NU1701 (*package does not specify its target platforms*).

To find fixes with NU1103 and NU1701, we allow unstable versions (denoted as V_{NU1103}) and versions that do not specify their target platforms (denoted as V_{NU1701}) in G. We have:

$$\max z_6 = -\sum_{v \in V_{\text{NU1103}}} v.\delta, \quad \max z_7 = -\sum_{v \in V_{\text{NU1701}}} v.\delta \tag{10}$$

It is challenging to find fixes with NU1608, because NU1608 means that some constraints in Equation 3 will be violated. Concretely, Equation 3 should be relaxed as:

$$\sum_{\substack{i \in V \\ i.\text{PKG}=p}} i.\delta \geq v.\delta. \tag{11}$$

This relaxation makes the model intractable because the search space explodes and we cannot trace the number of violated constraints in Equation 3. To address this problem, we introduce an auxiliary binary variable $x_{v,p}$ for each $v \in V \backslash v^*$ and $p \in \{v'.\text{PKG} | (v, v') \in E\}$. This variable indicates whether or not v's dependency constraint on p is violated. Then we have another relaxation for Equation 3:

$$x_{v,p} + \sum_{\substack{(v,i)\in E,\\ i.\text{PKG}=p}}^{i\in V} i.\delta \geq v.\delta. \tag{12}$$

We want to violate constraints in Equation 3 as few as possible:

$$\max z_8 = -\sum_{v,p} x_{v,p} \tag{13}$$

Similar to z (Equation 9), the overall objective function of the variant model for seeking risky fixes is:

$$\max z' = \sum_{k=1}^{8} \alpha'_k * z_k \tag{14}$$

4.5 Tuning Hyperparameters

We use Bayesian optimization to tune the hyperparameters $\boldsymbol{\alpha}$ in our models: $\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_5)$ for safe fixes, and $\boldsymbol{\alpha} = (\alpha'_1, ..., \alpha'_8)$ for risky fixes. Bayesian optimization is a global optimization algorithm built upon Bayesian inference and Gaussian process, that attempts to find the maximum/minimum value of a *black-box* function in as few iterations as possible. Here *black-box* means that we do not have access to the gradient of our models with respect to $\boldsymbol{\alpha}$.

To start a Bayesian optimization task, we need to specify the function f to be optimized. Intuitively, we wish to find an $\boldsymbol{\alpha}$ that makes the fixes returned by our model as similar as possible to real fixes provided by developers. Concretely, consider a training dataset $\mathcal{D}$ consisting of some pairs in the form of $(G, \hat{V})$: G is a DM graph, and $\hat{V}$ is the real fix (this training dataset is detailed in Sec 5.3). Then we have:

$$\min f(\boldsymbol{\alpha}) = \sum_{(G,\hat{V})\in\mathcal{D}} (z^*(G, \boldsymbol{\alpha}) - z(G, \hat{V}, \boldsymbol{\alpha})) \tag{15}$$

, where $z^*(G, \boldsymbol{\alpha})$ is the maximum value of the objective function z that our model solves (given G and $\boldsymbol{\alpha}$).

5 EVALUATION

We study two research questions in our evaluation section:

- **RQ3 (Effectiveness of NuFix):** *How effective is NuFix in fixing DM issues for .NET projects?*
- **RQ4 (Usefulness of NuFix)** *Do the fixes generated by NuFix make sense from practitioners' perspectives?*

To answer RQ3, we constructed a high-quality benchmark, to evaluate NuFix's capability of fixing DM issues.

To answer RQ4, we invited ten experienced .NET experts from industry to manually validate the fixes generated by NuFix, and further reported our generated fixes to the corresponding project developers in open-source community for validation.

5.1 Benchmark Construction

We considered the 2,500 .NET projects in $subjectSet_2$ as subjects. By mining these projects' code commits on GitHub, we constructed a high-quality benchmark consisting of real commits for resolving DM issues. Specifically, we filtered commits based on four criteria:

- Commits with ≤ 10 dependency changes were filtered out.

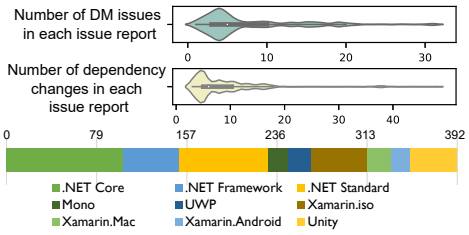

Figure 6: Statistics of 392 instances in our benchmark (log scale)

Table 1: Statistics of 392 documented fixes in our benchmark

	NU1605	NU1107	NU1202	NU1203	NU1108
N_{IR}	192	61	365	12	6
Max	39	5	24	3	1
Avg	2.66	1.79	3.67	0.07	0.02

Table 2: Statistics our training and test datasets

Training Dateset			
# Fixes inducing warnings† (39)			# Fixes not inducing warnings
NU1608	NU1103	NU1701	91
17	26	4	
Test Dataset			
# Fixes inducing warnings† (62)			# Fixes not inducing warnings
NU1608	NU1103	NU1701	200
36	57	7	

†Some fixes induce more than one types of warnings.

- We only kept the commits whose logs explicitly stated that the revisions aimed to upgrade/migrate the projects' platforms. This is based on our empirical finding that platform changes often trigger more complicated DM issues.
- For each project, we manually upgraded/migrated the project to the specified new platform. Only commits which could reveal DM issues were kept.

The final benchmark consists of 392 $\langle Input, Fix \rangle$ instances:

- *Input* is a 4-tuple $\langle C, P_0, P_1, R \rangle$. Specifically, C is a set of packages paired with their specified versions, extracted from the problematic configuration file before merging the commit; P_0 is the original .NET platform before updating; P_1 is the target .NET platform after merging the commit; R is the constraints of package versions. Note that, to be fair in comparison with real fixes, R excludes all package versions that were released after the timestamp of code commit.
- *Fix* is a set of dependency changes. We should not directly use all changed dependencies of the commit as *Fix*, because not all these changes are to resolve the DM issues. As such, we define *Fix* as the minimum subset of dependency changes in a code commit, satisfying that the subset can tackle the DM issue in P_1.

As shown in Figure 6 and Table 1, the instances in benchmark cover five types of DM issues and involve the projects targeting nine types of platforms. Each instance concerns 8.2 DM issues on average. The number of dependency changes concerned in each fix varies from 3 to 50 (i.e., 8.4 ± 5.9).

5.2 RQ3: Effectiveness of NuFix

We briefly describe the experiment design and results in this section. To ease presentation, we refer to the fixes derived by NuFix as *N-fix*, and a fix derived by developers in our benchmark as a *real fix*.

Data Splits. According to approaches [32, 43, 53, 55, 68], a sample size ten times the number of hyperparameters is reasonable for training. Since NuFix requires 13 hyperparameters, we randomly heldout 130 of the 392 instances as the training dataset for tuning

Table 3: Tuned hyperparameters used in our object functions

Seeking safe fixes					
$\boldsymbol{\alpha} = (\alpha_1, ..., \alpha_5) = (0.73, 0.14, 0.12, 0.01, 0.03)$					
$\lvert\alpha_1{*}z_1\rvert$	$\lvert\alpha_2{*}z_2\rvert$	$\lvert\alpha_3{*}z_3\rvert$	$\lvert\alpha_4{*}z_4\rvert$	$\lvert\alpha_5{*}z_5\rvert$	
2.19	1.36	1.23	0.81	0.78	
Seeking risky fixes					
$\boldsymbol{\alpha} = (\alpha'_1, ..., \alpha'_8) = (0.06, 0.01, 0.01, 4{*}10^{-4}, 1.2{*}10^{-3}, 0.01, 0.08, 0.83)$					
$\lvert\alpha'_1{*}z_1\rvert$	$\lvert\alpha'_2{*}z_2\rvert$	$\lvert\alpha'_3{*}z_3\rvert$	$\lvert\alpha'_4{*}z_4\rvert$	$\lvert\alpha'_5{*}z_5\rvert$	$\lvert\alpha'_6{*}z_6\rvert$	$\lvert\alpha'_7{*}z_7\rvert$	$\lvert\alpha'_8{*}z_8\rvert$
0.20	0.19	0.17	0.07	0.06	0.02	0.02	0.15

hyperparameters. The rest of them are regarded as the test dataset. Statistics of two datasets are described in Table 2.

Evaluation Metrics. We define four evaluation metrics below:

- *FR* (Fixing Ratio): the proportion of DM issue cases for which the technique can derive fixes.
- *SFR* (Safely Fixing Ratio): the proportion of DM issue cases for which the technique can derive safe fixes.
- *TPR* (Test Passing Ratio): the proportion of derived fixes that can pass the projects' bundled tests.
- *Similarity*: how similar are the derived fixes to the corresponding real fixes. We define *Similarity* based on *IoU*:

$$Similarity = \frac{|Changes_N \cap Changes_R|}{|Changes_N \cup Changes_R|} \quad (16)$$

where $Changes_N$ and $Changes_R$ are the changes induced by the derived and real fix, respectively. We define two types of similarity based on how the equivalence operator between two dependency changes: (a) *Type A*: two changes will be regarded equivalent, if they share the same package, major version, and direct dependency constraints; (b) *Type B*: two changes will be regarded equivalent, if they share the same package and major version. For each instance in our benchmark, we averaged the *Similarity* values of the top $k = 5$ derived fixes as the result.

Comparison. Two baseline approaches are considered in our study:

- ***CUDF for .NET***: CUDF [26] is proposed to resolve the package upgradeability problem for GNU/Linux-based distributions using a linear programming model. We adapt it to derive DM issue fixes (denoted as *C-fixes*) by encoding NuGet's build rules. It defines five user preferences to explore configurations. We only consider three of them: *fewer newly installed packages*, *fewer changes in direct dependencies* and *fewer packages that are not up-to-date*, while the rest two preferences are not applicable to NuGet's build rules. In CUDF, the weights of user preferences are not rigorously defined. As such, we adopt the empirical weights assigned for the above three user preferences in approach [26]: 1, 1,$|V|$+1, where $|V|$ is available version count of required packages.
- ***PyDFix for .NET***: PyDFix [67] is proposed to fix unreproducibility in Python builds caused by dependency version errors. We adopt it to derive fixes for DM issues (denoted as *P-fixes*) based on NuGet's build rules. PyDFix considers the available versions of packages that cause DM issues as the search space of fixing strategies. Such strategies are sorted by the highest version priority. It iteratively tries out each fixing strategy and then adds it into a documented fix if such a strategy can repair at least one unresolved DM issue. This procedure continues until all the DM issues are resolved, or all the fixing strategies have been tested. Note that in each iteration, it re-builds the project to verify the fixing strategy. Since such a process is computationally expensive, we set PyDFix's time budget to be 120 minutes.

Table 4: NuFix's effectiveness on DM issue fixing

	FR			*SFR*				*TPR*		
	NuFix	*CUDF*	PyDFix	NuFix	*CUDF*	PyDFix	*Benchmark*	NuFix	*CUDF*	PyDFix
Difficult	100%	60.2%	0.0%	60.2%	60.2%	0.0%	43.9%	69.4%	32.2%	0.0%
Moderate	100%	87.9%	28.8%	87.9%	87.9%	22.7%	67.4%	75.3%	35.6%	15.1%
Easy	100%	87.5%	65.6%	87.5%	87.5%	53.1%	71.9%	76.1%	38.0%	31.2%
Average	100%	77.5%	41.9%	77.5%	77.5%	33.5%	59.2%	73.3%	34.6%	20.8%
Time (Avg)	NuFix: 12.5 seconds			*CUDF*: 10.2 seconds				PyDFix: 115.4 minutes		

	Similarity					
	NuFix		*CUDF*		PyDFix	
	Type A	*Type B*	*Type A*	*Type B*	*Type A*	*Type B*
Difficult	50.1%	68.2%	18.5%	38.8%	0.0%	0.0%
Moderate	57.5%	76.2%	23.4%	39.8%	13.5%	14.2%
Easy	64.9%	85.1%	20.6%	35.3%	25.4%	32.5%
Average	55.7%	74.3%	21.2%	38.9%	16.8%	19.8%

†#Difficult cases (N_s: >=8): 98; #Moderate cases (N_s: 4–8): 132; #Easy cases (N_s: <4): 32

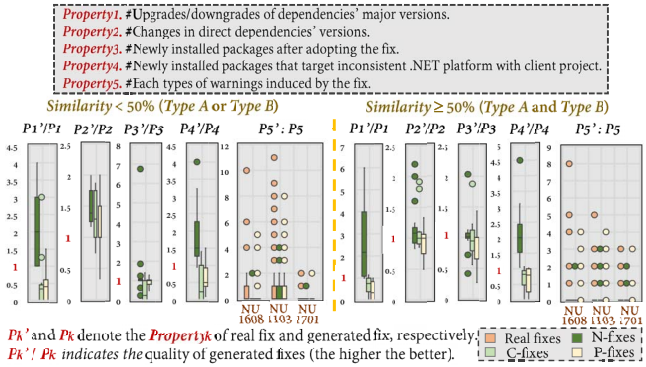

Figure 7: Distribution of five properties of N-fixes and real fixes

Hyperparameters. Table 2 shows the tuned hyperparameters, by applying Bayesian optimization algorithm to our training dataset. To understand how NuFix balances the developers' preferences, we compared the average value of each weighted preference-based factor (i.e., $\alpha_k{*}z_k$ and $\alpha'_k{*}z_k$ for our two optimization models, respectively), which indicates the factor influence for seeking fixes.

From Table 3, we can tell that for .NET practitioners: (1) ensuring the compatibility of packages (indicated by $k = 1, 2, 3$) plays a leading role in DM issue fixing; In practice, dependency upgrades ($k = 2$) are preferable to downgrades ($k = 3$); (2) NU1608 warnings (indicated by $k = 8$) are much more intolerable than NU1103 and NU1701 warnings for developers; (3) developers indeed prefer to install packages as few as possible ($k = 4$) and introduce fewer packages targeting inconsistent .NET platform with their client projects ($k = 5$), but these two preferences are not the top priority.

Results. Table 4 shows the evaluation results. To ease discussion, we further divided our testing dataset into three groups based on their difficulties (by #dependency changes N_s in real fixes). NuFix generates fixes for each .NET project within 12.5 seconds and achieves a 100% *FR* for the instances with different difficulties. 77.5% N-fixes are safe fixes, of which the safely fixing ratio is higher than real fixes (59.2%). While CUDF achieves $FR = SFR = 77.5\%$, since it does not support a variant model to seek risky fixes if safe fixes do not exist. In contrast, PyDFix exhibits poor performance in our evaluation. The main reason is: DM issues in .NET ecosystem is much more complicated than those in Python ecosystem (on average 2.4 dependency changes for fixing unreproducibility in a Python build [67] **v.s.** 8.4±5.9 dependency changes in a real fix of our benchmark), thus the iterative search methodology of PyDFix is insufficient to deal with the exponentially large search space.

For the 192 out of 262 instances having bundled test cases, we deployed their CI tools to further evaluate whether N-fixes would

cause test failures. Encouragingly, NuFix achieves 73.3% *TPR* on average, owing to its well-defined objective function (i.e., prefer fewer upgrades/downgrades of dependencies' major versions and fewer changes in direct dependency versions). The above *TPR* can be improved via iteratively refining the search scope of package versions by developers. However, CUDF and PyDFix do not restrict their algorithms to seek solutions with fewer package upgrades/downgrades crossing major versions. Instead, they reward the fixes with more up-to-date packages. As result, on average, they achieve 34.6% and 20.8% *TPR*, respectively.

NuFix achieves the *Type A* and *Type B* similarities of 55.7% and 74.3%, respectively, which significantly outperforms the baseline approaches. Figure 7 shows the distribution of five properties of the generated fixes. We can observe that: (1) N-fixes can better satisfy *Preferences* 1 and *2* than real fixes, which is indicated by the fact that P_1'/P_1 and P_2'/P_2 are significantly greater than 1; (2) For NuFix, *Preference 3* is a secondary factor comparing to *Preference 1 and 2*, as P_3'/P_3 varies from 0 to 7; (3) For CUDF, most of P_2'/P_2 and P_3'/P_3 are greater than 1 and close to those of N-fixes, since it considers *Properties 2* and *3* as user preferences. (4) CUDF and PyDFix prefer to install newest package versions and does not take *Properties 1* and *4* into account. As such, its P_1'/P_1 and P_4'/P_4 are significantly lower than those of N-fixes. (5) P_5 and P_5' indicate that N-fixes induced far fewer build warnings than C-fixes, P-fixes and real fixes; (6) Low *Similarity* does not necessarily mean that N-fixes are not good, as the above phenomena are more significant when the *Similarity* is low. To further understand these N-fixes with low *Similarity*, we performed expert validation in § 5.3.

5.3 RQ4: Usefulness of NuFix

In this research question, we aim to validate the usefulness of NuFix by means of the feedback from .NET experts and project developers.

Setup for Expert Validation. A DM issue can have multiple fixing solutions. As such, we want to evaluate if a fix generated by NuFix is valid even if it differs significantly from the real fix. To conduct the evaluation, we invited .NET experts from industry to manually assess the generated fixes.

- *Participants.* We invited ten .NET experts with over three years development experience from Microsoft to participate in our study.
- *Subjects.* All the top fixes generated by NuFix in § 5.2 were selected if (1) their *Similarity* (either *Type A* or *B*) with the corresponding real fixes is less than 50%; and (2) they do not result in test failures. 123 pairs of N-fixes and real fixes were selected.
- *Procedure.* The formal study session was supervised by two authors. We provided a 10-minute briefing to the ten participants on the two tasks before starting the tasks.
  - *Task 1. Background information collection.* We collected from the participants (1) their experience in years, (2) the .NET platforms that they have worked on, and (3) whether they have encountered DM issues in their projects in the past three months. We collected the information to investigate whether the participants had experience in dealing with DM issues.
  - *Task 2. Fix assessment.* Each participant was assigned to examine 12 or 13 fix pairs. The participant was asked to choose the preferred fix in each fix pair. To facilitate that, the participant was given the original dependency configuration as well as the DM issues arose for each fix pair. We also provided each participant a visualization of the package dependency graph after adopting a fix. The graph contains the updated packages and the information of their five properties (Figure 7) as auxiliary info. To avoid biased results, we did not disclose to participants which of the fix in a fix pair is the real fix or the N-fix.

Setup for Project Developer Validation. We further validated the usefulness of N-fixes by checking if the project developers would replace their real fixes with N-fixes. We wanted to focus on the N-fixes of those projects whose developers were more likely to give us feedback. Therefore, from the 123 N-fixes (selected in expert validation), we chose the ones if they satisfy: (1) their code repositories have commit records in the past month (actively maintained), and (2) there are no dependency updates since these real fixes are merged (ensuring the current configurations equal the real fixes). Following these two criteria, we finally chose 25 N-fixes. We made pull requests (PRs) to revise their configurations based on the N-fixes. The pull requests also explained the improvements of our suggested configurations in terms of the five properties (e.g., inducing fewer packages).

Ethical Considerations. We provided all .NET experts with a form that informed them about the study purpose, the data we collected and stored, and email to contact us in case they had concerns. Additionally, before making PRs to project developers for validation, we have thoroughly tested the fixes derived by NuFix, to avoid spamming the open-source community. All PRs were submitted in compliance with projects' contribution guidelines and licenses.

Results of Expert Validation. Figures 8(1) and (2) show the experience background of our participants. 40% of participants have 10-15 years experience in .NET development and 70% of them are proficient with more than four types of platforms. In particular, they all have experience with DM issues in the past three months.

The expert validation results are described in Figure 8(3). In 70.7% of the cases, participants preferred the fixes generated by NuFix to the real fixes. While in 22.8% of cases, they confirmed that either of the two fixes was acceptable for them, since the fixes were evenly matched in practitioners' desired properties. In particular, an expert commented on a pair of fixes for resolving DM issues (2 NU1107 and 12 NU1202 errors) in commit#*3bb49c4* of `h5` [20]: "*One fix (real fix) addressed the issues at the cost of inducing two NU1608 warnings. While the other fix (N-fix) is excellent because it did not cause any warnings with one more upgrade of package's major version.*"

In eight cases (6.5%), experts only adopted the real fixes. We further analyzed these instances and found that to guarantee fewer upgrades/downgrades of dependencies' major versions (higher weight in NuFix's objective function), NuFix derived the fixes at the expense of introducing numerous additional packages. For example, to resolve eight NU1202 issues in commit#*008e928* of project `Restaurant` [21], developers changed four direct dependencies' major versions in the real fix. While NuFix fix only upgrades two dependencies' major versions, it induces 75 more packages.

Results of Project Developer Validation. Among our 25 submitted PRs that suggested project developers to replace their real fixes with the ones derived by NuFix, 20 PRs (80.0%) have been quickly merged. Most of the fixes are confirmed by the affected

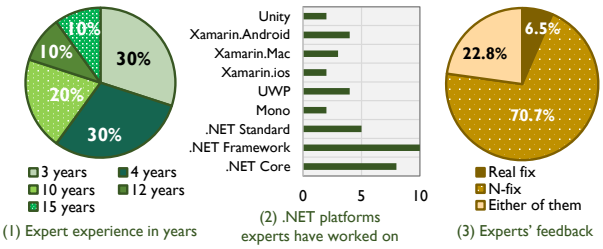

Figure 8: Results of expert validation

projects such as `Dropbox` [19]. The remaining 5 PRs are pending. Developers of these projects have not made decisions. We have not received rejection of our suggestions. The results indicate that N-fixes can resolve the encountered DM issues better than or as good as the original fixes. A developer specially left a comment in PR#43 [18] of `charlessolar`: "*At the time configuration had issues, but I believe that they were fixed by these commits.*"

Developers also showed interest in the NuFix tool. For example, one developer commented "*For my own curiosity, what is the name of the tool you used to generate the fixes? Please do share it once you are ready!*", in `Fortnox`'s PR #41 [17]. Details of our PRs can be found on NuFix's website.

6 THREATS TO VALIDITY

Subjectivity of Inspection. In our empirical study, we adopted manual analysis to construct the taxonomy of fixing strategies and discuss developers' configuration preferences. The manual analysis may pose threats to the validity of empirical findings. To minimize this threat, we adopted an open coding procedure [33], to categorize the results. Each issue report was inspected and labelled by two co-authors independently. Any discrepancy was discussed and resolved by the third author, until consensus was reached.

Bias in Expert Validation. The data from expert validation may be subject to bias. To limit this bias, we investigated participants' profiles and ensured they have over three years .NET development experience. In particular, they have dealt with DM issues in the past three months. To guarantee that they fully understand the tasks, we gathered ten experts and gave them a 10-minute briefing at the beginning of study session. We did not disclose to participants which fix was the real fix/N-fix in each given fix pair.

7 RELATED WORK

Build-Failure Repair. Studies on automated program repair [27, 28, 35, 40, 42, 44–46, 50, 51, 54, 56, 58, 60, 61, 66, 69, 71, 73, 74, 78, 86–90] are emerging in recent years, especially the build-failure repair techniques [29, 47, 62–64, 67, 80, 92]. For example, Bart [80] is a tool that summarizes *Maven* build logs and provides relevant online links to reduce developers' resolution efforts. HireBuild [47] uses historical build fix information to generate fixes for build scripts that have led to similar types of build failures. HoBuff [62] applies dataflow analysis to build logs and generating fixes for the faults analyzed. BuildMedic [64] repairs dependency-related build failures for Maven projects based on three common fixing strategies. PyDFix [67] is the latest technique proposed to detect and fix unreproducibility in Python builds caused by dependency version errors. BuildMedic [64] and PyDFix [67] share a common workflow: they considered the possible package version updates as the search space of fixing strategies. They iteratively try out each fixing strategy and then add it into a documented fix if such a strategy can repair at least one unresolved build error. This procedure continues until all the build errors are resolved, or all the fixing strategies have been tested. Such a process is computationally expensive since it re-builds the project in each iteration to verify the fixing strategy. However, the search space of possible fixes for DM issues in .NET ecosystem is exponentially large. Our comparison results (see § 5.2) shows that the above iterative search methodology exhibits poor performance in repairing DM issues.

Dependency solving. Various studies have been made on dependency management [24, 31, 34, 36–39, 41, 48, 52, 57, 59, 72, 79, 91]. Studies [49, 70, 83–85] mainly focus on detecting dependency issues rather than fixing. In this paper, we focus on resolving dependency issues arising from software evolution. Approaches [25, 26, 30, 65, 75, 77, 81, 82] aim to resolve the package upgradeability problem for GNU/Linux-based distributions. Vouillon et al. [81, 82] identify a set of packages that cannot be installed/upgraded together, using graph-theoretic transformations. Mancinelli et al. [65] propose an approach that formulates the resolution of dependencies and inter-package conflicts as a linear optimization problem among feasible upgrade solutions. Recent techniques [26, 30, 75] make further enhancement in the consideration of developer preferences.

However, these existing works are inapplicable to resolve the DM issues in .NET projects because they lack: (1) empirical supports for the defined user preferences, (2) a rigorous process to assign weights for user preferences, and (3) providing an optimal sub-solution when a perfect solution fixing all issues does not exist. In this paper, we address these limitations and propose an effective technique to derive DM issue fixes, in which developers' preferences are empirically investigated and systematically explored.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we empirically studied DM issues in .NET projects, which are prevalent and challenging for developers to address. We investigated their fixing strategies and developers' configuration preferences. To help developers repair DM issues, we developed the NuFix tool, by formulating this task as a binary integer optimization problem. The evaluation confirmed the effectiveness of NuFix, as its derived fixes were in line with developers' desired properties. In the future, we plan to use effective testing/static analysis techniques to efficiently validate NuFix's generated fixes.

ACKNOWLEDGMENTS

The authors express thanks to the anonymous reviewers for their constructive comments. This work was conducted during the first and second authors' visit at MSRA in 2021 (Microsoft Research Asia StarTrack Program). The work is supported by the National Natural Science Foundation of China (Grant Nos. 62141210, 61932021, 61902056), the Hong Kong RGC/GRF grant 16205821, MSRA grant, ITF grant (MHP/055/19, PiH/255/21), Shenyang Young and Middle-aged Talent Support Program (Grant No. ZX20200272), and Open Fund of State Key Lab. for Novel Software Technology, Nanjing University (KFKT2021B01).

REFERENCES

[1] 2021. Interactive compatibility matrix of .NET Standard. https://docs.microsoft.com/en-us/dotnet/standard/net-standard#net-implementation-support. (2021). Accessed: 2021-06-01.

[2] 2021. Introduction for .NET platforms in Microsoft documentation. https://docs.microsoft.com/en-us/dotnet/standard/components. (2021). Accessed: 2021-06-01.

[3] 2021. Issue #1039 of project refit. https://github.com/reactiveui/refit/issues/1039. (2021). Accessed: 2021-06-01.

[4] 2021. Issue #11 of project Unmockable. https://github.com/riezebosch/Unmockable/issues/11. (2021). Accessed: 2021-06-01.

[5] 2021. Issue #1139 of project EasyNetQ. https://github.com/EasyNetQ/EasyNetQ/issues/1139. (2021). Accessed: 2021-06-01.

[6] 2021. Issue #18 of project ServiceFabric. https://github.com/autofac/Autofac.ServiceFabric/issues/18. (2021). Accessed: 2021-06-01.

[7] 2021. Issue #81 of project Installer. https://github.com/dotnet/installer/issues/81. (2021). Accessed: 2021-06-01.

[8] 2021. Issue #886 of project AzureAD. https://github.com/AzureAD/azure-activedirectory-identitymodel-extensions-for-dotnet/issues/886. (2021). Accessed: 2021-06-01.

[9] 2021. Issue#14393 of project aspnetcore. https://github.com/dotnet/aspnetcore/issues/14393. (2021). Accessed: 2021-06-01.

[10] 2021. Issue#153 of project AndroidX. https://github.com/xamarin/AndroidX/issues/153. (2021). Accessed: 2021-06-01.

[11] 2021. Issue#36550 of project corefx. https://github.com/dotnet/corefx/pull/36550. (2021). Accessed: 2021-06-01.

[12] 2021. Issue#425 of project GoogleCloudPlatform. https://github.com/GoogleCloudPlatform/dotnet-docs-samples/issues/425. (2021). Accessed: 2021-06-01.

[13] 2021. Issue#545 of project standard. https://github.com/dotnet/standard/issues/545. (2021). Accessed: 2021-06-01.

[14] 2021. Issue#5696 of project aspnetcore. https://github.com/dotnet/aspnetcore/issues/5696. (2021). Accessed: 2021-06-01.

[15] 2021. Issue#972 of project EasyNetQ. https://github.com/EasyNetQ/EasyNetQ/issues/972. (2021). Accessed: 2021-06-01.

[16] 2021. NuGet central repository. https://www.nuget.org/. (2021). Accessed: 2021-02-01.

[17] 2021. PR #41 of project fortnox.NET. https://github.com/zenta-ab/fortnox.NET/pull/41. (2021). Accessed: 2021-06-01.

[18] 2021. PR#43 of charlessolar. https://github.com/charlessolar/eShopOnContainersDDD/pull/43. (2021). Accessed: 2021-06-01.

[19] 2021. Project Dropbox. https://github.com/dropbox/dropbox-sdk-dotnet. (2021). Accessed: 2021-06-01.

[20] 2021. Project h5. https://github.com/theolivenbaum/h5. (2021). Accessed: 2021-02-01.

[21] 2021. Project Restaurant. https://github.com/chayxana/Restaurant-App. (2021). Accessed: 2021-02-01.

[22] 2021. python-mip. https://pypi.org/project/mip/. (2021). Accessed: 2021-02-01.

[23] 2021. Semantic versioning strategy. https://semver.org/. (2021). Accessed: 2021-06-01.

[24] Pietro Abate and Roberto Di Cosmo. 2011. Predicting upgrade failures using dependency analysis. In *2011 IEEE 27th International Conference on Data Engineering Workshops*. IEEE, 145–150.

[25] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 547–551.

[26] Pietro Abate, Roberto Di Cosmo, Ralf Treinen, and Stefano Zacchiroli. 2012. Dependency solving: a separate concern in component evolution management. *Journal of Systems and Software* 85, 10 (2012), 2228–2240.

[27] Christoffer Quist Adamsen, Anders Møller, Rezwana Karim, Manu Sridharan, Frank Tip, and Koushik Sen. 2017. Repairing event race errors by controlling nondeterminism. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 289–299.

[28] Afsoon Afzal, Manish Motwani, Kathryn Stolee, Yuriy Brun, and Claire Le Goues. 2019. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering* (2019).

[29] Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. 2014. Fault localization for build code errors in makefiles. In *Companion Proceedings of the 36th International Conference on Software Engineering*. 600–601.

[30] Josep Argelich, Daniel Le Berre, Inês Lynce, Joao Marques-Silva, and Pascal Rapicault. 2010. Solving Linux upgradeability problems using boolean optimization. *LoCoCo 2010: Logics for Component Configuration* (2010).

[31] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2015. How the apache community upgrades dependencies: an evolutionary study. *Empirical Software Engineering* 20, 5 (2015), 1275–1317.

[32] Yaile Caballero, Rafael Bello, Alberto Taboada, Ann Nowe, Maria M Garcia, and Gladys Casas. 2006. A new measure based in the rough set theory to estimate the training set quality. In *2006 Eighth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE, 133–140.

[33] John W. Creswell. 2013. *Qualitative Inquiry and Research Design: Choosing Among Five Approaches (3rd Edition)*. SAGE Publications, Inc.

[34] Soto-Valero César, Harrand Nicolas, Monperrus Martin, and Baudry Benoit. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empirical Software Engineering* 26, 45 (2021).

[35] Brett Daniel, Tihomir Gvero, and Darko Marinov. 2010. On test repair using symbolic execution. In *Proceedings of the 19th international symposium on Software testing and analysis*. 207–218.

[36] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us About Semantic Versioning? *IEEE Transactions on Software Engineering* 47, 6 (2021), 1226–1240.

[37] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the Evolution of Technical Lag in the npm Package Dependency Network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 404–414.

[38] Alexandre Decan, Tom Mens, and Eleni Constantinou. 2018. On the evolution of technical lag in the npm package dependency network. In *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 404–414.

[39] Alexandre Decan, Tom Mens, and Philippe Grosjean. 2019. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering* 24, 1 (2019), 381–416.

[40] Favio DeMarco, Jifeng Xuan, Daniel Le Berre, and Martin Monperrus. 2014. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*. 30–39.

[41] Jens Dietrich, David Pearce, Jacob Stringer, Amjed Tahir, and Kelly Blincoe. 2019. Dependency Versioning in the Wild. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 349–359.

[42] Rui-Zhi Dong, Xin Peng, Yi-Jun Yu, and Wen-Yun Zhao. 2013. Requirements-driven self-repairing against environmental failures. In *2013 International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 241–244.

[43] Rosa L Figueroa, Qing Zeng-Treitler, Sasikiran Kandula, and Long H Ngo. 2012. Predicting sample size required for classification performance. *BMC medical informatics and decision making* 12, 1 (2012), 1–10.

[44] Bernd Fischer, Ando Saabas, and Tarmo Uustalu. 2009. Program repair as sound optimization of broken programs. In *2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. IEEE, 165–173.

[45] Patrick Franz, Thorsten Berger, Ibrahim Fayaz, Sarah Nadi, and Evgeny Groshev. 2021. ConfigFix: Interactive Configuration Conflict Resolution for the Linux Kernel. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. IEEE, 91–100.

[46] Zebao Gao, Zhenyu Chen, Yunxiao Zou, and Atif M Memon. 2015. Sitar: Gui test script repair. *Ieee transactions on software engineering* 42, 2 (2015), 170–186.

[47] Foyzul Hassan and Xiaoyin Wang. 2018. Hirebuild: An automatic approach to history-driven repair of build scripts. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 1078–1089.

[48] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *2018 IEEE/ACM 40th International Conference on Software Engineering: New Ideas and Emerging Technologies Results (ICSE-NIER)*. IEEE, 101–104.

[49] Kaifeng Huang, Bihuan Chen, Bowen Shi, Ying Wang, Congying Xu, and Xin Peng. 2020. Interactive, effort-aware library version harmonization. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 518–529.

[50] Ishtiaque Hussain and Christoph Csallner. 2010. Dynamic symbolic data structure repair. In *2010 ACM/IEEE 32nd International Conference on Software Engineering*, Vol. 2. IEEE, 215–218.

[51] Li Jia, Hao Zhong, Xiaoyin Wang, Linpeng Huang, and Xuansheng Lu. 2021. The Symptoms, Causes, and Repairs of bugs inside a Deep Learning Library. *The Journal of Systems Software* (2021), to appear.

[52] Zhouyang Jia, Shanshan Li, Tingting Yu, Chen Zeng, Erci Xu, Xiaodong Liu, Ji Wang, and Xiangke Liao. 2021. DepOwl: Detecting Dependency Bugs to Prevent Compatibility Failures. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. 86–98.

[53] HM Kalayeh and David A Landgrebe. 1983. Predicting the required number of training samples. *IEEE transactions on pattern analysis and machine intelligence* 6 (1983), 664–667.

[54] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. 2013. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering* 39, 11 (2013), 1597–1610.

[55] Marc D Kohli, Ronald M Summers, and J Raymond Geis. 2017. Medical image data and datasets in the era of machine learning—whitepaper from the 2016 C-MIMI meeting dataset session. *Journal of digital imaging* 30, 4 (2017), 392–399.

[56] Anil Koyuncu, Kui Liu, Tegawendé F Bissyandé, Dongsun Kim, Martin Monperrus, Jacques Klein, and Yves Le Traon. 2019. iFixR: bug report driven program repair. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 314–325.

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

[58] Xuan Bach D Le, David Lo, and Claire Le Goues. 2016. History driven program repair. In *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, Vol. 1. IEEE, 213–224.

[59] Li Li, Daoyuan Li, Tegawendé F Bissyandé, Jacques Klein, Haipeng Cai, David Lo, and Yves Le Traon. 2017. Automatically locating malicious packages in piggybacked android apps. In *2017 IEEE/ACM 4th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. IEEE, 170–174.

[60] Xiangyu Li, Marcelo d'Amorim, and Alessandro Orso. 2019. Intent-preserving test repair. In *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 217–227.

[61] Xuliang Liu and Hao Zhong. 2018. Mining StackOverflow for program repair. In *Proc. SANER*. 118–129.

[62] Yiling Lou, Junjie Chen, Lingming Zhang, Dan Hao, and Lu Zhang. 2019. History-driven build failure fixing: how far are we?. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 43–54.

[63] Yiling Lou, Zhenpeng Chen, Yanbin Cao, Dan Hao, and Lu Zhang. 2020. Understanding build issue resolution in practice: symptoms and fix patterns. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 617–628.

[64] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 106–117.

[65] Fabio Mancinelli, Jaap Boender, Roberto Di Cosmo, Jerome Vouillon, Berke Durak, Xavier Leroy, and Ralf Treinen. 2006. Managing the complexity of large free and open source package-based software distributions. In *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 199–208.

[66] Martin Monperrus, Simon Urli, Thomas Durieux, Matias Martinez, Benoit Baudry, and Lionel Seinturier. 2019. Repairnator patches programs automatically. *Ubiquity* 2019, July (2019), 1–12.

[67] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing Dependency Errors for Python Build Reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2021)*. 439–451.

[68] Sayan Mukherjee, Pablo Tamayo, Simon Rogers, Ryan Rifkin, Anna Engle, Colin Campbell, Todd R Golub, and Jill P Mesirov. 2003. Estimating dataset size requirements for classifying DNA microarray data. *Journal of computational biology* 10, 2 (2003), 119–142.

[69] Emerson Murphy-Hill, Thomas Zimmermann, Christian Bird, and Nachiappan Nagappan. 2014. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering* 41, 1 (2014), 65–81.

[70] Jibesh Patra, Pooja N Dixit, and Michael Pradel. 2018. Conflictjs: finding and understanding conflicts between javascript libraries. In *Proceedings of the 40th International Conference on Software Engineering*. 741–751.

[71] Xuhong Ren, Bing Yu, Hua Qi, Felix Juefei-Xu, Zhuo Li, Wanli Xue, Lei Ma, and Jianjun Zhao. 2020. Few-Shot Guided Mix for DNN Repairing. In *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 717–721.

[72] César Soto-Valero, Amine Benelallam, Nicolas Harrand, Olivier Barais, and Benoit Baudry. 2019. The Emergence of Software Diversity in Maven Central. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. 333–343.

[73] Shin Hwei Tan, Jooyong Yi, Sergey Mechtaev, Abhik Roychoudhury, et al. 2017. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 180–182.

[74] Haoye Tian, Kui Liu, Abdoul Kader Kaboré, Anil Koyuncu, Li Li, Jacques Klein, and Tegawendé F Bissyandé. 2020. Evaluating Representation Learning of Code Changes for Predicting Patch Correctness in Program Repair. In *The 35th IEEE/ACM International Conference on Automated Software Engineering (ASE 2020)*.

[75] Paulo Trezentos, Inês Lynce, and Arlindo L Oliveira. 2010. Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In *Proceedings of the IEEE/ACM international conference on Automated software engineering*. 427–436.

[76] Andrew Troelsen. 2003. *C# and the. NET Platform*. Vol. 56. Springer.

[77] Chris Tucker, David Shuffelton, Ranjit Jhala, and Sorin Lerner. 2007. OPIUM: Optimal Package Install/Uninstall Manager. In *29th International Conference on Software Engineering (ICSE'07)*. 178–188.

[78] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2019. Learning how to mutate source code from bug-fixes. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 301–312.

[79] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. 2018. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the PyPI ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 644–655.

[80] Carmine Vassallo, Sebastian Proksch, Timothy Zemp, and Harald C Gall. 2018. Unbreak my build: Assisting developers with build repair hints. In *2018 IEEE/ACM 26th International Conference on Program Comprehension (ICPC)*. IEEE, 41–4110.

[81] Jérôme Vouillon and Roberto Di Cosmo. 2013. On Software Component Co-Installability. *ACM Trans. Softw. Eng. Methodol.* 22, 4, Article 34 (2013), 35 pages.

[82] Jérôme Vouillon and Roberto Di Cosmo. 2013. Broken Sets in Software Repository Evolution. In *Proceedings of the 2013 International Conference on Software Engineering*. 412–421.

[83] Ying Wang, Ming Wen, Yepang Liu, Yibo Wang, Zhenming Li, Chao Wang, Hai Yu, Shing-Chi Cheung, Chang Xu, and Zhiliang Zhu. 2020. Watchman: monitoring dependency conflicts for Python library ecosystem. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 125–135.

[84] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 319–330.

[85] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I have a stack trace to examine the dependency conflict issue?. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 572–583.

[86] Chu-Pan Wong, Priscila Santiesteban, Christian Kästner, and Claire Le Goues. 2021. VarFix: balancing edit expressiveness and search effectiveness in automated program repair. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 354–366.

[87] Meng Wu, Shengjian Guo, Patrick Schaumont, and Chao Wang. 2018. Eliminating timing side-channel leaks using program repair. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 15–26.

[88] Jooyong Yi, Umair Z Ahmed, Amey Karkare, Shin Hwei Tan, and Abhik Roychoudhury. 2017. A feasibility study of using automated program repair for introductory programming assignments. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 740–751.

[89] Jooyong Yi, Shin Hwei Tan, Sergey Mechtaev, Marcel Böhme, and Abhik Roychoudhury. 2018. A correlation study between automated program repair and test-suite metrics. *Empirical Software Engineering* 23, 5 (2018), 2948–2979.

[90] Tingting Yu and Michael Pradel. 2018. Pinpointing and repairing performance bottlenecks in concurrent programs. *Empirical Software Engineering* 23, 5 (2018), 3034–3071.

[91] Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús González-Barahona. 2018. An empirical analysis of technical lag in npm package dependencies. In *International Conference on Software Reuse*. Springer, 95–110.

[92] Chen Zhang, Bihuan Chen, Linlin Chen, Xin Peng, and Wenyun Zhao. 2019. A large-scale empirical study of compiler errors in continuous integration. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 176–187.