SMARTPIP: A Smart Approach to Resolving Python Dependency Conflict Issues

Chao Wang[1], Rongxin Wu[1], Haohao Song[1], Jiwu Shu[1], Guoqing Li[2]
[1]School of Informatics, Xiamen University, Xiamen, China
[2]Xiamen Meiya Pico Information Co., Ltd., Xiamen, China
wangc@stu.xmu.edu.cn,wurongxin@xmu.edu.cn
songhaohao2021@stu.xmu.edu.cn,jwshu@xmu.edu.cn,ligq@300188.cn

ABSTRACT

As one of the representative software ecosystems, PyPI, together with the Python package management tool PIP, greatly facilitates Python developers to automatically manage the reuse of third-party libraries, thus saving development time and cost. Despite its great success in practice, a recent empirical study revealed the risks of dependency conflict (DC) issues and then summarized the characteristics of DC issues. However, the dependency resolving strategy, which is the foundation of the prior study, has evolved to a new one, namely the backtracking strategy. To understand how the evolution of this dependency resolving strategy affects the prior findings, we conducted an empirical study to revisit the characteristics of DC issues under the new strategy. Our study revealed that, of the two previously discovered DC issue manifestation patterns, one has significantly changed (*Pattern A*), while the other remained the same (*Pattern B*). We also observed, the resolving strategy for the DC issues of *Pattern A* suffers from the efficiency issue, while the one for the DC issues of *Pattern B* would lead to a waste of time and space. Based on our findings, we propose a tool SMARTPIP to overcome the limitations of the resolving strategies. To resolve the DC issues of *Pattern A*, instead of iteratively verifying each candidate dependency library, we leverage a pre-built knowledge base of library dependencies to collect version constraints for concerned libraries, and then convert the version constraints into the SMT expressions for solving. To resolve the DC issues of *Pattern B*, we improve the existing virtual environment solution to reuse the local libraries as far as possible. Finally, we evaluated SMARTPIP in three benchmark datasets of open source projects. The results showed that, SMARTPIP can outperform the existing Python package management tools including PIP with the new strategy and CONDA in resolving DC issues of *Pattern A*, and achieve 1.19X - 1.60X speedups over the best baseline approach. Compared with the built-in Python virtual environment (VENV), SMARTPIP reduced 34.55% - 80.26% of storage space and achieved up to 2.26X - 6.53X speedups in resolving the DC issues of *Pattern B*.

ASE '22, October 10–14, 2022, Rochester, MI, USA

ACM ISBN 978-1-4503-9475-8/22/10…$15.00
https://doi.org/10.1145/3551349.3560437

CCS CONCEPTS

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

KEYWORDS

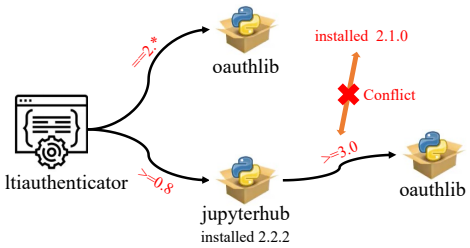
Python, dependency conflict, dependency resolving

ACM Reference Format:
Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, Guoqing Li. 2022. SMARTPIP: A Smart Approach to Resolving Python Dependency Conflict Issues. In *37th IEEE/ACM International Conference on Automated Software Engineering (ASE '22), October 10–14, 2022, Rochester, MI, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3551349.3560437

1 INTRODUCTION

Modern software development typically relies on software ecosystem to facilitate the third-party libraries installation [24, 27, 39]. A successful example is the Python ecosystem, PyPI [12], which provides a central repository to store a large number of Python third-party libraries [37]. By March 2022, it had indexed nearly 3 million release versions of libraries, each of which is described with metadata (e.g., library name, version information, and dependency on other libraries) [28]. To import a library hosted in PyPI, developers need to declare version constraints (i.e., the constraints that the versions of libraries must satisfy) to restrict the set of compatible library versions in configuration files, such as *setup.py* and *requirements.txt* [22, 26, 32, 41]. Then, complying with the version constraints, the python library installer, PIP, will automatically install the imported library and other libraries that are depended on this imported library, in a recursive manner. PIP relieves developers of the heavy burden of managing library dependencies. However, due to the complex library dependencies, the automatic installation process would come with the risk of Dependency Conflict (DC) issues, when multiple version constraints declared for the same library conflict with each other, leading to build failures.

Figure 1 shows a real-world DC issue example: *issue #21* [4] from the open source project *ltiauthenticator*. As shown in *ltiauthenticator*'s configuration file, it directly depends on *jupyterhub* whose version constraint is *"jupyterhub >= 0.8"* and *oauthlib* whose version constraint is *"oauthlib == 2.∗"*. Note that, the default installation policy of PIP is to download the latest version in PyPI that satisfies the version constraints [8]. Thus, in Figure 1, the two latest versions of libraries, *jupyterhub-2.2.2* and *oauthlib-2.1.0*, will be installed. However, for *jupyterhub-2.2.2*, its direct dependency *oauthlib* whose version constraint is *"oauthlib >= 3.0"* conflicts with the prior installed library version *oauthlib-2.1*, which causes




```
pip (legacy resolver) install log:
ERROR: jupyterhub 2.2.2 has requirement oauthlib>=3.0,
 but you'll have oauthlib 2.1.0 which is incompatible.
pip (backtracking resolver) install log:
Collecting jupyterhub>=0.8
  Downloading jupyterhub-2.2.2-py3-none-any.whl (3.7 MB)
  Downloading jupyterhub-2.2.1-py3-none-any.whl (3.7 MB)
  Downloading jupyterhub-2.2.0-py3-none-any.whl (3.7 MB)
  Downloading jupyterhub-2.1.1-py3-none-any.whl (3.7 MB)
  ......
  Downloading jupyterhub-0.9.6-py3-none-any.whl (3.1 MB)
```

Figure 1: Illustrative examples of DC issues.

a DC issue. When using PIP with the version prior to *20.3*, it will report a build failure as shown in Figure 1.

To understand the characteristics of the Python DC issues, Wang et al. [41] conducted the first empirical study on the manifestation patterns and fixing strategies of DC issues from the Python open source projects, and proposed a technique named WATCHMAN to diagnose the DC issues. However, since PIP 20.3 (released after the publication of WATCHMAN), the dependency resolving strategy, which is the foundation of the prior empirical study [41], has evolved to a new one, namely the backtracking dependency resolving strategy [13]. For ease of the explanation, in this work, the dependency resolving strategy used in the version prior to PIP 20.3 was referred to as **the legacy strategy**, while the one used after PIP 20.3 was referred to as **the backtracking strategy**. The evolution of the dependency resolving strategy would significantly change the manifestations of the DC issues. For the DC issue shown in Figure 1, instead of getting stuck in the situation where *oauthlib-2.1.0* is incompatible with *"oauthlib >= 3.0"*, PIP with the backtracking strategy will continue to search other candidates and eventually take *jupyterhub-0.9.6* as the final choice. Since *jupyterhub-0.9.6* has no version constraint on *oauthlib*, the DC issue is solved. The aforementioned case indicates that, the evolution of the dependency resolving strategies would significantly affect the prior research findings, as well as the necessity of the existing DC issue diagnosis technique. Therefore, we conducted an empirical study to revisit the characteristics of the Python DC issues.

Our empirical study is designed based on the two manifestation patterns of Python DC issues discovered in the prior study [41]: conflicts caused by remote dependency updates (*Pattern-A*) and conflicts affected by the local environment (*Pattern-B*). Note that, both DC issue patterns will lead to build failure using PIP with the legacy strategy. The root cause for the DC issues of *Pattern-A* is that the updates of some remote dependency (in PYPI) change the version constraints of the concerned libraries. The root cause for the DC issues of *Pattern-B* is that the required version constraints of a remote dependency are incompatible with the locally installed tools or libraries. To understand the differences between the two different dependency resolving strategies we design the following two research questions.

- **RQ1:** Will the DC issues of *Pattern-A* manifest the same under the two different dependency resolving strategies? Do fixing strategies for the DC issues of *Pattern-A* need to be adapted under the new dependency resolving strategy?
- **RQ2:** Will the DC issues of *Pattern-B* manifest the same under the two different dependency resolving strategies? Do fixing strategies for the DC issues of *Pattern-B* need to be adapted under the new dependency resolving strategy?

Through investigating the research questions, we have made several interesting findings. First, the backtracking strategy resolves the majority of the DC issues of *Pattern-A*, but there is still room to improve the efficiency of finding the satisfiable candidate versions of the libraries of concern. Moreover, the DC issues of *Pattern-A* that cannot be resolved are mainly due to the performance issue of the backtracking strategy and the unsatisfiable version constraints. Second, the backtracking strategy behaves the same as the legacy one when encountering the DC issues of *Pattern-B*. Although creating an isolated virtual environment is a feasible fixing strategy, it would result in a waste of time and space, since a significant number of compatible common libraries are downloaded and stored repeatedly in different virtual environments.

To address the limitations observed in our empirical study, we propose a technique SMARTPIP. To improve the efficiency of the backtracking strategy in resolving DC issues of *Pattern-A*, we construct an offline dependency knowledge base, which persists the direct dependencies information (including version constraints) for all the libraries, collected and updated from the Python ecosystem. Then, we resort to the offline dependency knowledge base to collect the version constraints of all the required libraries, and then convert the version constraints into the SMT expressions for solving. SMARTPIP is more efficient than PIP with the backtracking strategy. This is because, the backtracking strategy requires to download and verify one candidate version of the concern libraries for each time, until all the dependency libraries satisfy the version constraints, which is highly time-consuming. Compared with that, the pre-built dependency knowledge base facilitates us to collect the version constraints without constantly downloading a number of candidate libraries. SMARTPIP is also more efficient than other Python package management tools that encode dependency resolving as an SAT problem, e.g., CONDA. This is because, to deal with a dependency library with n candidate versions, SMARTPIP converts the version constraints into the SMT expressions with the time complexity of $O(1)$, while CONDA converts them into the SAT expressions with the time complexity of $O(n^2)$. To resolve the DC issues of *Pattern-B*, in SMARTPIP, we design a new solution to use the virtual environment. First, to share the libraries among all the virtual environments, we first create a local repository that hosts all the downloaded libraries, and then create a soft link pointing to a desired library in the corresponding virtual environment. Second, to reuse the libraries in the local repository, we design an algorithm to integrate the version information of these local libraries into the version constraints, so that the local libraries are given a higher priority to be selected. In this way, SMARTPIP can greatly reduce the time to download the required libraries and the space to store them.

To evaluate SMARTPIP, we used three datasets, HG2.9K [29], WATCHMAN DC issues dataset [9] and AAAI 2021 dataset [10]. To evaluate the effectiveness and efficiency of resolving the DC issues of *Pattern-A*, we compared SMARTPIP with two representative Python package management tools, PIP with the backtracking strategy and CONDA. The results showed that, SMARTPIP can resolve all of DC issues and achieve 1.19X - 1.60X speedup over the best baseline technique. To evaluate the effectiveness of resolving the DC issues of *Pattern-B*, we used SMARTPIP and PIP with VENV (a built-in Python virtual environment) to build all the Python projects in three datasets. The results showed that, the time speedup and space reduction of SMARTPIP over PIP with *venv* are 2.26X - 6.53X and 34.55% - 80.26% respectively.

In summary, the contributions of this paper are as follows:

- We performed an empirical study to revisit the characteristics of the Python DC issues under the new dependency resolving strategy. Our findings help understand the impact of the new strategy in two DC issue patterns observed in the prior study and their fixing strategies. We also observed some new technical limitations of the new strategy.
- Based on the knowledge learned from our empirical study, We proposed SMARTPIP to overcome the limitations of the strategy in resolving the DC issues.
- We evaluated the proposed technique SMARTPIP using the three benchmark datasets of the real-world Python projects. The evaluation results show that, SMARTPIP can effectively resolve the two different patterns of DC issues using less time and space.

2 PRELIMINARIES

To facilitate the management of library dependencies, various Python package managers, such as PIP and CONDA, are proposed and deployed. The existing Python package managers provide diverse functionalities. In this section, we introduce only two of these functionalities, i.e., dependency resolving and virtual environment management, since they are related to DC issues.

2.1 Dependency Resolving

To use a library in the ecosystem, a Python project needs to specify version constraints of the desired libraries. The process of figuring out the appropriate version for each desired library that satisfies the specified version constraints is referred to as "dependency resolving". Based on how the version constraints are solved, we classify the existing Python package managers into two categories.

The first category follows an iterative manner to compute a feasible solution. Specifically, for each time, it first searches and downloads one version of the desired library L_i which satisfies the version constraints, and then iteratively verifies whether the version constraint on each direct dependency library of L_i is satisfied. PIP, with either the legacy strategy or the backtracking strategy, POETRY[20], and PIPENV are the representative tools of this category.

The second category encodes the version constraints of the desired libraries into a SAT solving problem, and then resorts to a solver to directly compute the appropriate version of each desired library. Essentially, this line of technique would not lead to the build failure caused by DC issues of *Pattern-A*, if there is a feasible solution to satisfy the version constraints of all the desired libraries. A representative tool of this category is CONDA [15]. However, due to the complexity of encoding dependency resolving to the SAT problem, CONDA has suffered from the performance issue for a long time, which has been extensively discussed in its issue reporting system [2, 5–7, 17]. Besides, as admitted by CONDA developers, CONDA is typically slower than PIP in dependency resolving [2].

2.2 Virtual Environment Management

The virtual environment enables to isolate the installation of libraries for different projects without affecting one another. As pointed out by a prior study [41], the DC issues of *Pattern-B* can be mitigated by creating the virtual environment. Despite the numerous Python package managers that provide the virtual environment management functionality, they essentially work in a similar manner. Specifically, PIP VENV (Python built-in virtual environment), PIPENV, and VIRTUALENV will treat each virtual environment individually, and download and install the require libraries without sharing and reusing the libraries from other virtual environments.

2.3 Scope Clarification

In this work, we limit the scope of our empirical study to the Python package manager PIP for the following reasons. First, PIP is one of the most popular and representative Python package managers. It also shares the same/similar dependency resolving and virtual environment management mechanism as most of other Python package managers, including POETRY, PIPENV, VIRTUALENV and so on. Thus, understanding the characteristics of DC issues with PIP is also meaningful for other Python package managers. Second, PIP manages the dependency libraries hosted in PyPI which is currently the largest Python ecosystem, and can be used for general Python projects. Meanwhile, some Python package managers are applicable in some specific types of projects, since they leverage a comparatively small-scale ecosystem. For example, CONDA maintains only 7,500+ libraries for Python [16], and typically limits the usage in data science projects. Therefore, using PIP enables us to conduct the empirical study on general Python projects.

3 EMPIRICAL STUDY AND MOTIVATION

The characteristics of DC issues for Python projects were first empirically studied by Wang et al. [41]. Their findings greatly inspired and facilitated the future studies on Python DC issues. However, their results were based on the dependency resolving strategy of PIP at the time of their study, which has been obsolete since PIP 20.3. Due to a generally limited understanding of how this evolution of PIP's dependency resolving strategy affects the manifestation patterns and fixing patterns of DC issues, we performed an empirical study on the same dataset of the prior study [41] by comparing the two different strategies.

3.1 The Design of Empirical Study

3.1.1 *Design for RQ1:*

To answer RQ1, we used the dataset of DC issues that are collected or reported by WATHCHMAN [41]. We manually checked the bug reports of the DC issues of *Pattern-A* as well as the ones reported in their subsequent submissions. Specifically, by inspecting the bug report and the *Pull Request* for fixing DC issues, we identified the version of the project before the target DC issue was fixed. Then,

Table 1: Statistics for constructing the *Pattern-B dataset*.

Group	# of project pairs	% of DC issues of *Pattern B*
0	47,401	/
[1, 5]	33,639	2.10% (706/33,639)
[6, 10]	11,496	5.88% (676/11,496)
[11, 20]	4,268	9.77% (417/4,268)
[21, 50]	2,765	21.88% (605/2,765)
[51, +∞)	431	24.13% (104/431)

we tried to reproduce the DC issues using the legacy dependency resolving strategy, and filtered out the ones which cannot be reproduced. These unreproducible cases are mainly due to the following reasons: (1) the code repository of the target Python project was removed; (2) the updates of some remote dependency in PyPI changed the version constraints of the concerned libraries and fixed the DC issues; or (3) the bug report did not contain necessary reproduction information (e.g., the version where the DC issue occurs, commits or pull requests for fixing DC issues). Finally, we gathered 82 DC issues from 117 Python projects. To ease the explanation, we refer to this dataset as "***Pattern-A dataset***".

3.1.2 Design for RQ2:

For RQ2, we cannot use the DC issues of *Pattern-B* discovered by the prior study [41]. This is because, the authors failed to provide the developers' real local environments [41], and thus these DC issues are impossible to reproduce. Therefore, to answer RQ2, we build a dataset of the DC issues of *Pattern-B* in a simulated way as follows. (1) We randomly select a pair of Python projects hosted in PyPI, and then compute the number of their common dependency libraries (without considering the version). (2) For each Python projects pair, we consider one to be the target python project to be installed, and the other one (together with its dependency libraries) as the local environment. We repeat the aforementioned steps 100,000 times, and eventually obtain 2,508 pairs of Python projects that can cause the DC issues of *Pattern-B*. To ease the explanation, we refer to this dataset as "***Pattern-B dataset***".

Although the *Pattern-B dataset* is not collected from the real-world developers' environments, it reflects the simplest scenario that can induce DC issues of *Pattern-B* to some extent. During constructing *Pattern-B dataset*, we made several interesting observations. (1) As shown in Table 1, by randomly selecting pairs of Python projects, 52.60% (52,599/100,000) of the project pairs share at least one common library. (2) The probability of inducing the DC issues of *Pattern-B* increases with the number of common libraries that a pair of Python projects share. As shown in Table 1, based on the number of common libraries, we categorize each pair into five groups: [1, 5], [6, 10], [11, 20], [21, 50], [51, +∞]. In the Group [1,5], the probability of inducing DC issues is around 2.1%, while this probability increases to 24.13% in the Group [51, +∞]. The observation is not surprising, since the DC issues have originated from the conflict about the version constraints of common libraries.

3.2 The Empirical Study Results

3.2.1 Answer to RQ1.

To answer RQ1, we apply the legacy and backtracking strategies to the *Pattern-A dataset*. In the following, we discuss the manifestation patterns and fixing strategies of the issues in detail with the illustrative examples.

Finding 1: *pip with the backtracking strategy can successfully resolve 70.73% (58 in 82) of the DC issues of Pattern-A. To complete the installation of all the libraries, for each project, the backtracking strategy requires to download and verify on average 20 redundant candidate versions, leading to the efficiency issue.*

In *Pattern-A dataset*, the backtracking strategy can solve most of the DC issues. As shown in Figure 1, the legacy strategy will get stuck in the installation of *jupyterhub-2.2.2*, since its direct dependency *oauthlib* whose version constraint is *"oauthlib >= 3.0"* in conflict with *oauthlib-2.1.0* which was previously installed. As shown in the installation log, the backtracking strategy will keep searching all the candidate versions of *jupyterhub*, i.e., the ones which satisfy the version constraints *"jupyterhub >= 0.8"*. When trying *jupyterhub-0.9.6*, its direct dependency *oauthlib* has no version constraints, and thus we can keep the previously installed *oauthlib-2.1.0*. The above steps are iterative, until the version constraints of all the dependency libraries are satisfied. As seen from the example, the backtracking strategy allows to iteratively download some candidate versions and discards them when the corresponding version constraints are later violated. This iterative search would result in the downloading of some redundant candidates. In the example of Figure 1, the backtracking strategy iteratively downloaded 19 versions of *jupyterhub* in total, of which each is more than 3MB. We surveyed all the projects that can be solved by the backtracking strategy in the *Pattern-A dataset*, and found that, on average, it requires to download 20 redundant candidate libraries for each project before completing the installation of all the required libraries.

Finding 2: *When the search space of the backtracking strategy is too large, it will terminate the search of the dependency libraries and lead to the failure of resolving the DC issues of Pattern-A.*

In terms of the cases where the backtracking strategy fails to resolve, one of the reasons is that the search space of the backtracking is too large. For example, when using the backtracking strategy to install the project *imgsync* [3], it reports an error message *"This is taking longer than usual. You might need to provide the dependency resolver with stricter constraints to reduce runtime"*. Moreover, The log message *"pip._vendor.resolvelib.resolvers.ResolutionTooDeep: 2000000"* indicates that, when the search depth is too large, to avoid downloading too many candidates pip decides to terminate the search. Although we only found one case of such failure in our study, essentially, this failure can be considered to be an extreme case of the efficiency issue pointed out by *Finding 1*.

Finding 3: *When version constraints of all the required libraries are unsolvable, neither the legacy strategy nor the backtracking strategy can resolve such DC issues. These DC issues can be fixed by changing the version constraints of the Python project or its dependency libraries.*

In *Pattern-A dataset*, there are 29.27% (24 in 82) projects that cannot be resolved by the two existing strategies of pip. Specifically, the legacy strategy will directly report an error whenever the version constraints of a concerned libraries is violated, while the backtracking strategy will exhaustively search all the possible candidate versions of the required libraries and then eventually report the error. Figure 2 shows such an example. *Client* directly depends on *netlib* of which the version constraint is *"netlib <= 0.11.1"* and *pyopenssl* of which the version constraint is *"pyopenssl == 0.13.1"*.

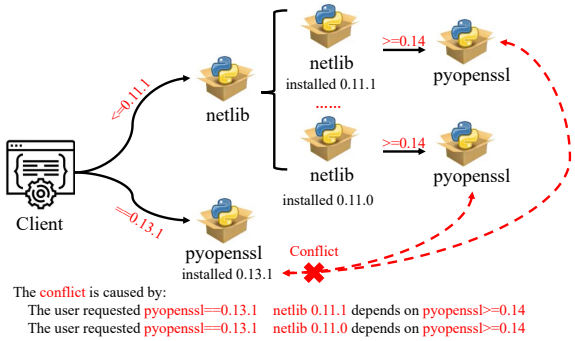


Figure 2: An example of unsolvable version constraints.

Table 2: The proportion of incompatible libraries.

Group	DC issues of *Pattern B*	# of Common libraries (*avg*)	% of incompatible in common libraries
[1, 5]	706	3.42	38.30% (1.31/3.42)
[6, 10]	676	7.21	21.50% (1.55/7.21)
[11, 20]	417	14.13	11.75% (1.66/14.13)
[21, 50]	605	26.59	10.91% (2.90/26.59)
[51, +∞)	104	59.37	4.19% (2.49/59.37)

However, all the feasible versions of *netlib* depend on *pyopenssl* and the version constraint is *"pyopenssl >= 0.14"*, which is always contradicted with *Client*'s requirements on *pyopenssl* and the version constraint is *"pyopenssl == 0.13.1"*. Essentially, such DC issues indicate that there is a contradiction among the version constraints of the project and its required dependency libraries. Therefore, it should notify the developers of the project or the dependency libraries to assist the fixing of such DC issues, which correspond to the fixing strategies summarized by the prior study WATCHMAN [41].

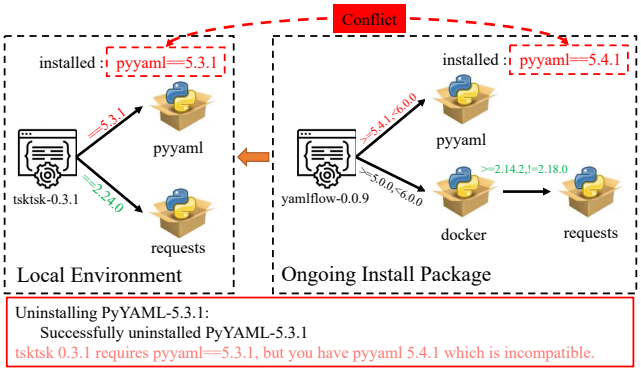
Answer to RQ1: The DC issues of *Pattern-A* that manifest as the build failures under the legacy strategy have been mostly resolved by the backtracking strategy. Meanwhile, a minority of DC issues of *Pattern-A*, whose version constraints of the dependency libraries are unsolvable, cannot be resolved by the backtracking strategy. Instead, solving these DC issues requires adjustment to the declaration of version constraints among the Python project and its dependency libraries, as suggested by the prior study [41].

3.2.2 *Answer to RQ2.*

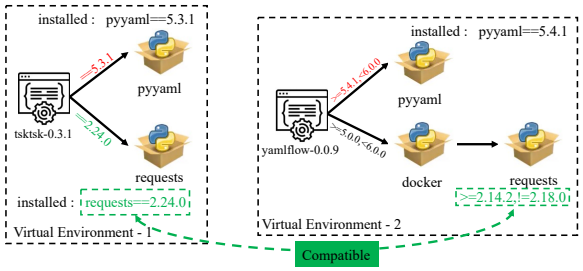
To answer RQ2, we applied both the backtracking strategy and the legacy strategy to the *Pattern-B dataset*.

Finding 4: *Both the backtracking strategy and the legacy strategy behave the same for the DC issues of the Pattern-B, and will uninstall the libraries of the local environment which are incompatible with the version constraints of the ongoing installed libraries.*

Figure 3(a) shows an example of DC issues of the *Pattern-B*. The local environment includes *tsktsk-0.3.1* and its dependency libraries *pyyaml-5.3.1* and *requests-2.24.0*. When installing *yamlflow-0.0.9*, the version constraint of its dependency library *pyyaml* is *"pyyaml >= 5.4.1 ∧ pyyaml < 6.0.0"*, which is conflicted with the local library *pyyaml-5.3.1*. In this case, both of the strategies will uninstall the local one, and install the latest version with respect to



(a) Installing *yamlflow-0.0.9* in the local environment.



(b) Create virtual environments for *tsktsk-0.3.1* and *yamlflow-0.0.9*.

Figure 3: An example of DC issues of *Pattern-B* and its fixing strategy.

the constraint *"pyyaml >= 5.4.1 ∧ pyyaml < 6.0.0"*, i.e., *pyyaml-5.4.1*. However, this replacement version of *pyyaml* may lead to the runtime error when running the local project *tsktsk-0.3.1* in future.

As discussed in a prior study [41], creating the virtual environment for each project is a feasible fixing strategy for resolving the DC issues of *Pattern-B*. Therefore, in this study, we use the Python built-in virtual environment, i.e., VENV, to isolate the installation of each project in the pairs. Our study confirms the feasibility of this solution under both the backtracking strategy and the legacy strategy. Meanwhile, we also made some interesting observations on the limitations of using the Python built-in virtual environment.

Finding 5: *Using Python's built-in virtual environment to isolate the installation of each project resolves the DC issues of Pattern-B. However, due to this isolation, it loses the chance to reuse and share the compatible common libraries between different virtual environments.*

Creating an isolating virtual environment to install projects brings the additional time and space cost. Figure 3(b) shows an example of creating virtual environments for two Python projects *tsktsk-0.3.1* and *yamlflow-0.0.9*, respectively. This resolves the DC issue shown in Figure 3(a). As we can see, both projects require the dependency library *requests*, and the version constraints are *"requests == 2.24.0"* and *"requests >= 2.14.2 ∧ requests! = 2.18.0"* respectively, which are compatible with each other. Thus, it is feasible to share the common library *requests-2.24.0* for the two projects. However, due to the isolated virtual environment, it needs to download *requests*−2.24.0 and *requests-2.27.1*, respectively, resulting in additional time and space cost.

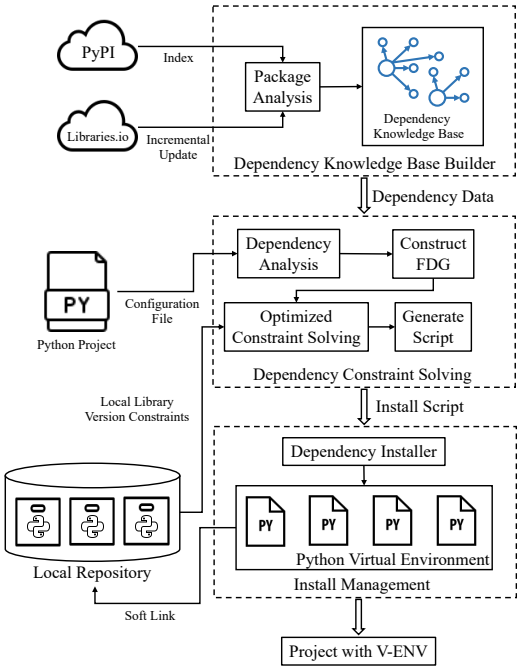


Figure 4: Architecture of SMARTPIP.

To further understand the potential of reusing the compatible common libraries, we investigated the logs of installing each pair of the project of the *Pattern-B dataset* in the same local environment. Note that, if the version constraints of a common library are incompatible between two projects, PIP will uninstall the previously installed version of this library. We analyzed the logs about such uninstallation, and computed the number of incompatible common libraries for each project pair. Table 2 shows the statistical results for the proportion of the incompatible common libraries between the project pairs in the *Pattern-B dataset*. The incompatible common libraries take a small proportion of the overall common libraries, ranging from 4.19% to 38.30%, and decrease with the increasing number of the common libraries. This indicates that, for 61.70% to 95.81% of common libraries, which are compatible ones in the installation of project pairs, are not reused and shared when creating the isolated virtual environments, leading to additional time and space cost.

Answer to RQ2: Both the backtracking strategy and the legacy strategy behave the same when resolving the DC issues of *Pattern-B*. Creating a virtual environment to isolate the installation of Python projects is a feasible fixing strategy. However, using the Python built-in virtual environment leads to a waste of time and space, since compatible common libraries are not shared or reused.

4 APPROACH

4.1 Overview

In this section, we described the SMARTPIP approach, which aims to solve the DC issues in Python projects. The overall structure of SMARTPIP is shown in Figure 4. SMARTPIP mainly includes an offline process and an online process. The offline process constructs the local dependency knowledge base for all the libraries hosted in

```
charset_normalizer~=2.0.0
idna>=2.5,<4
urllib3>=1.21.1,<1.27
certifi>=2017.4.17
```

(a) An example of *requirement.txt*.

```
setuptools.setup(
    name="Lib-A",
    version="0.0.1",
    long_description=long_description,
    package_dir={"": "src"},
    install_requires=['requests>=2.20.0'],
    ......
)
```

(b) An example of *setup.py*.

Figure 5: Examples of Python configuration files.

PyPI (See Section 4.2). The online process generates the installation script for a given Python project based on its configuration file (See Section 4.3), and finally manages the installation of all the required libraries (See Section 4.4).

4.2 Dependency Knowledge Base Building

As shown in the empirical study, PIP with the backtracking strategy is required to iteratively download one candidate version of a concerned library to obtain the version constraints of its direct dependency libraries. As we can expect, it would be inefficient when the number and the size of iterative downloaded libraries are large. To overcome this limitation, we propose building a local dependency knowledge base to store the version constraints of all the dependency libraries hosted in the PyPI, so that the version constraints can be collected without downloading the corresponding libraries.

4.2.1 Collection of Library Dependency Data.
PyPI is a central repository for third-party libraries in Python language, which is used in most of the Python project development. Therefore, our local dependency knowledge base was constructed by crawling from PyPI. Specifically, we developed a crawler to download all the Python projects in PyPI and the corresponding dependency configuration files. Due to the large number of libraries (around 3 million release versions), it took us three days to download all of them. Since PyPI will continually update, it is also important to keep the local knowledge base updated. However, downloading the PyPI repository each time from scratch is impractical since it would not be able to keep pace with the update of PyPI. To resolve this issue, we resort to *Libraries.io*, a platform that can monitor the library updates from 32 different library managers including PyPI. Based on that, we can incrementally crawl the Python libraries that have been updated since our last crawling, as well as their dependency configuration files. This can greatly reduce the time cost of maintaining the local dependency knowledge base.

4.2.2 Dependency Configuration Analysis.
After getting all the Python libraries from PyPI, we analyze each library's dependency configuration files to extract the version constraints of its dependency libraries. Typically, there are two types of dependency configuration files: *requirements.txt* and *setup.py*.

Analyzing *requirements.txt* is comparatively easy, as each line directly represents the version constraint of a direct dependency library. Figure 5(a) shows an example, which includes the version constraints of four dependency libraries.

Analyzing *setup.py* is sightly complicated. The version constraints of the direct dependency libraries are essentially one of the parameters (i.e., *install_requires*) used in the function call to a specific API

setuptools.setup. Figure 5(b) shows an example of *setup.py*. As the value of *install_requires* can be defined in various ways, statically analyzing its value would be unsound. To resolve this issue, we instrument *setup.py* by inserting a function which is identical to the API *setuptools.setup* and will print out the value of *install_requires*. By dynamically executing this instrumented program, this inserted function will intercept the original API *setuptools.setup* and print out the value of *install_requires*. In this way, we can soundly obtain the version constraints of the direct dependency libraries.

To facilitate SMT solver to understand the version constraint of each dependency library, we convert each version constraint (e.g., *oauthlib* == 2.∗) into an SMT expression using the following steps. First, for the library that is concerned with the given version constraint (e.g., *oauthlib*), we query the knowledge base to obtain the ordered list of all of its versions in the ecosystem. Note that, for each library, during the knowledge base building, we order all of its versions following PEP 440 (Python Enhancement Proposal 440) [1] which is the official Python version scheme for identifying and ordering library versions, and then associate each version with its index in the ordered list. Second, we follow PEP 440 to identify the versions that satisfy the given version constraint, and then leverage the index range of these versions to construct SMT expression. Take "*oauthlib* == 2.∗" as an example. There are 48 versions of *oauthlib* which have been ordered and indexed during the knowledge base building. We identify 9 versions of *oauthlib*, i.e., "2.0, 2.1, ..., 2.1.0", which satisfy the version constraint. Since the index of these versions range from 35 to 43, we then convert "*oauthlib* == 2.∗" into "$35 \leq oauthlib \leq 43$".

4.2.3 Using Dependency Knowledge Base In the Users' Site.
To save on users' time cost of building and updating the knowledge base, we provide users with the services of downloading and updating the pre-built knowledge base data which is constructed in a remote server. To compactly store the data for all the libraries in PyPI, the data structure of the knowledge base is elaborately designed. For example, we use a unique hash value to represent the name of each library, and use an index value to represent each version of a library. In our implementation, it only takes up 100 MB to store the pre-built knowledge base. Such compact data is affordable for use in the users' site. At the users' site, a local process will be independently started for loading the downloaded knowledge base and fetching the updated data periodically (e.g., once a day). In this way, we mitigate the computation cost of constructing and updating knowledge base in users' site.

4.3 Dependency Constraint Solving

In this section, we mainly introduce how to collect the version constraints of all the required libraries with respect to a given Python project. We also explain how to optimize the constraint solving to reuse the libraries in the local repositories.

4.3.1 Construct SMT expression.
Given a new Python project to install, we can resort to the same approach as described in Section 4.2.2 to analyze its dependency configuration files to extract its direct dependency libraries with their version constraints. Based on the direct dependency libraries, we then leverage the local dependency knowledge base to collect all the transitive dependency libraries with their version constraints.

Algorithm 1 Collect version constraint

```
Input: C : Project Configuration
Output: Expr : All version constraints of the required libraries
1:  /** L : Get all direct libraries with version constraints from configuration file.**/
2:  L ← getDirectLibraries(C)
3:  Expr ← TRUE
4:  /** L_i : Library name.
5:  VC_i : Library version constraints.**/
6:  for each tuple < L_i, VC_i > ∈ L do
7:      S_i ← ∅
8:      Expr_{L_i} ← libraryAnalysis(L_i, VC_i, S_i)
9:      Expr ← AND(Expr, Expr_{L_i})
10: function libraryAnalysis(L_i, VC_i, S)
11:     if L_i not in S then
12:         add L_i into S
13:         /** Convert version constraints to version list
14:         by using dependency knowledge base.**/
15:         V ← analysisVersionConstraint(VC_i)
16:         Expr_V ← FALSE
17:         for each version V_i ∈ V do
18:             /** D : Get all direct libraries
19:             with version consistent from library.**/
20:             D ← getDirectLibraries(V_i)
21:             Expr_D ← TRUE
22:             for each tuple < D_i, VC_{D_i} > ∈ D do
23:                 Expr_{D_i} ← libraryAnalysis(D_i, VC_{D_i}, S_i)
24:                 Expr_D ← AND(Expr_D, Expr_{D_i})
25:             /** Add expr for each version with OR.**/
26:             Expr_V ← OR(Expr_V, Expr_D)
27:         remove L_i from S
28:         Return AND(Expr_V, VC_i)
29:     /** If L_i has been resolved,
30:     only add its version constraint into SMT.**/
31:     Return VC_i
```

The pseudo-code of constructing the version constraints of the required libraries is shown in Algorithm 1. By analyzing the dependency configuration file, we obtain all the direct dependencies with their version constraints (Line 2, Algorithm 1). The SMT expression of the version constraints *Expr* is initialized with value *TRUE*. Then, we enumerate each direct dependency library L_i, collect the SMT expression $Expr_{L_i}$ for L_i and the L_i's required dependency libraries, and then perform the conjunction operation between $Expr_{L_i}$ and *Expr* for each time (Line 6-9, Algorithm 1). The function LibraryAnalysis takes a dependency library name L_i, the L_i's version constraint VC_i, and a set of library names S as the inputs. LibraryAnalysis returns the SMT expression of the version constraint for a given library L_i and L_i's required dependency libraries. The library name set s, which is one of the inputs of LibraryAnalysis, is used to avoid the computation of previously collected library's SMT expression.

4.3.2 Constraint Solving Optimization.
After collecting SMT expressions for the version constraints for the required libraries, SMARTPIP resorts to Z3-solver, the most popular and open source constraint solving tool [25, 36], to solve the constraints.

As pointed out by our empirical study (See Section 2), the Python built-in virtual environments do not share or reuse the common compatible versions of libraries, leading to additional time and space cost. To achieve the reuse of the local libraries, we propose to give a higher priority to those versions of the local libraries. Specifically, we transform one version of the local library into an SMT expression each time, use the "AND" operation to integrate this SMT expression into the SMT expression collected in the prior

Algorithm 2 Optimizing version constraints

Input: $Expr$: SMT expression
Input: LR : all libraries in local repository
Output: L : list of library with version constraint (e.g., $requests == 2.20.0$)

```
1:  /** get all libraries name from Expr.**/
2:  LN ← getAllLibraries(Expr)
3:  for each library LN_i ∈ LN do
4:      if LN_i in LR then
5:          /** get all versions of the library
6:          in the local repository.**/
7:          V ← getAllVersion(LR, LN_i)
8:          for each version V_i ∈ V do
9:              /** save a copy.**/
10:             Expr_copy ← Expr
11:             Expr ← AND(Expr, LN_i == V_i)
12:             if Expr is sat then
13:                 break
14:             else
15:                 /** roll-back Expr. **/
16:                 Expr ← Expr_copy
```

step (Section 4.3.1), and then use Z3 to check whether they are satisfiable. If so, we keep this version of the local library in the SMT expression, and repeat the prior step until there no available local libraries. If not, we discard this version of local library, and repeat the prior step. In this way, we can optimally reuse the versions of the local libraries.

The pseudo-code of the above constraint solving optimization step is shown in Algorithm 2. It takes the input *Expr* which is computed by Algorithm 1, and the other input *LR* which is the list of the libraries downloaded in the local repository (See Section 4.4). SMARTPIP retrieves the list of library names from *Expr* (Line 2, Algorithm 1), and then enumerate each of them LN_i. If the local repository contains a list of versions for LN_i, it enumerates each version V_i, and checks whether the SMT expression which integrates $LN_i == V_i$ into *Expr* with "AND" operation is satisfiable. If satisfiable, we will preserve the version V_i using the constraint $LN_i == V_i$.

Note that, the installation policy of PIP is to select the latest version that satisfies the version constraints and avoid selecting the pre-release version [8]. To align with these preferences, for each library, we add the constraints to exclude its pre-release versions and instruct the SMT solver to compute the feasible solution that favors the maximum value (indicating the latest version).

4.4 Library Installation Management

As pointed out by our empirical study, Python built-in virtual environment creates one copy of each required library independently, and thus loses the chances to share the libraries from other virtual environments. To overcome this limitation, we introduce a local repository to store the downloaded libraries required by different virtual environments, and leverage the soft link to share them.

To resolve the DC issues of *Pattern-B*, SMARTPIP also resorts to the Python virtual environment to install the Python project. Note that, installing different versions of a same library is not allowed in the same Python virtual environment. Therefore, to facilitate the store and reuse of all the downloaded Python dependency libraries for different virtual environments, we create a local repository which does not belong to any virtual environment. The first-layer directory of the local repository is organized based on each library.

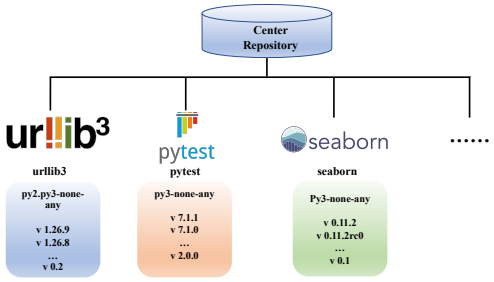


Figure 6: An example of central repository.

For different versions of the same library, SMARTPIP creates the second-layer directory to store each of them. Figure 6 shows an example of our local repository. When the dependency constraint solving process provides a compatible version of a concerned library, e.g., *urllib3-1.26.9*, SMARTPIP will first search whether a first-layer directory *urllib3* and its second-layer directory *v1.26.9* exist in the local repository. If so, SMARTPIP creates a soft link to point to it. If not, SMARTPIP will create the two-layer directories based on the predefined organization, download the given version of the library, store the downloaded library in the created directory, and eventually create a soft link in the virtual environment to point to the created directory. In this way, SMARTPIP can save the hard disk space, since each version of one library is only downloaded and stored once and can be accessed via soft links among different virtual environments.

The dependency constraint solving step delivers a feasible solution for the compatible versions of the required libraries. SMARTPIP then converts this feasible solution into a dependency configuration file that PIP can use to install the required libraries. Figure 7 shows an example of converting a feasible solution to an installation script. For the project *Client*, the solution of the compatible versions of the required libraries is *Lib-A 1.0*, *Lib-B 2.0*, *Lib-C 3.0*, and *Lib-D 4.0*. SMARTPIP converts these library versions into the equations *Lib-A==1.0*, *Lib-B==2.0*, *Lib-C==3.0*, *Lib-D==4.0* in the dependency configuration file, i.e., "smartpip.txt". Using the dependency configuration file, SMARTPIP checks whether the required libraries have already been in the local repository, and then generates the corresponding installation script which will exclude the download of the previously downloaded libraries. In the aforementioned example, since *Lib-B 2.0* and *Lib-D 4.0* have been in the local repository, SMARTPIP only downloads the other two required libraries in the local repository, and creates the soft link to point to the directory of the local repository in the virtual environment.

4.5 Technical Comparison with CONDA

CONDA encodes the dependency resolving as an SAT problem (See Section 2.1), while our approach SMARTPIP encodes it as an SMT problem. The difference between SMARTPIP and CONDA for encoding the dependency resolving problem is the key that SMARTPIP can mitigate the performance issue of CONDA.

Let us take an illustrative example. Suppose that there is a dependency library L_i with the version constraint $v_1 \leq L_i \leq v_n$, and the ordered list of the versions that satisfy this version constraint is $v_1, v_2, ..., v_n$. CONDA needs to encode each version v_i with the values 1 and 0 to indicate whether v_i is selected or not. Since the selection of v_i is mutually exclusive with other versions, selecting v_i is

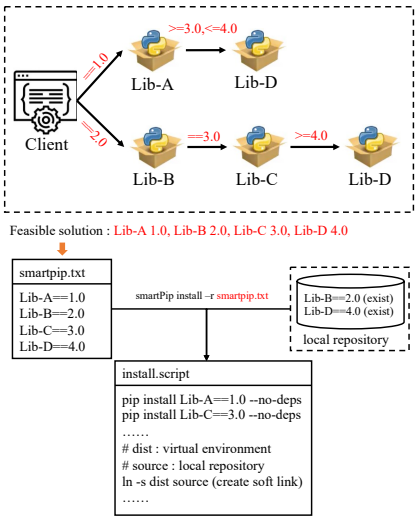


Figure 7: Convert a feasible solution to an installation script.

encoded as an SAT expression $S_{L_i} = (\neg v_1 \wedge \neg v_2 \wedge ... \wedge v_i \wedge ... \wedge \neg v_n)$. Since $v_1, v_2, ..., v_i, ..., v_n$ are candidate versions for a feasible solution, CONDA needs to encode it as $S_{L_1} \vee S_{L_2} \vee ... \vee S_{L_i} \vee ... \vee S_{L_n}$. SMARTPIP can greatly simplify the problem encoding. As the index of the versions that satisfy the version constraint range from 1 to n, the version constraint is converted to an SMT expression $1 \leq L_i \leq n$. Essentially, in the above example, CONDA constructs the SAT expression with the time complexity of $O(n^2)$, while SMARTPIP constructs the SMT expression with the time complexity of $O(1)$. Due to this time complexity reduction, SMARTPIP can be much more efficient than CONDA.

5 EVALUATION

5.1 Experimental Setup

We evaluate the effectiveness of SMARTPIP in resolving the DC issues of *Pattern-A* and *Pattern-B* using real-world open source Python projects against the following two research questions.

- **RQ3:** Can SMARTPIP resolve the DC issues of *Pattern-A* successfully? How much time can SMARTPIP save, compared to PIP with the backtracking strategy and CONDA?
- **RQ4:** When resolving the DC issues of *Pattern-B*, how much time and space can SMARTPIP save, compared to PIP with the Python built-in virtual environment?

To study **RQ3** and **RQ4**, we select the three publicly available datasets of real-world open source Python projects from the existing research studies.

- WATCHMAN dataset [9]: This dataset contains the Python projects whose DC issues were detected by WATCHMAN. This dataset was also used in Section 3. Note that, as shown in Section 3.2.1, the DC issues in 58 out of 82 projects are solvable, and are thus used in the evaluation.
- HG2.9K dataset [29]: This dataset contains the Python projects which were used to evaluate the effectiveness of the Python dependency inference techniques [30, 31, 40]. We obtained 36 Python projects which contain DC issues.

Table 3: The Success Rate Comparison.

Dataset	Conda	PIP - backtracking	SMARTPIP
WatchMan	6/58 (10.3%)	57/58 (98.3%)	58/58 (100%)
HG2.9K	8/36 (22.2%)	36/36 (100%)	36/36 (100%)
AI	2/26 (7.7%)	26/26 (100%)	26/26 (100%)
Sum	16/120 (13.3%)	119/120 (99.2%)	120/120 (100%)

- AAAI 2021 dataset: This dataset contains the implementations for the research papers published in AAAI 2021, a top-tier conference in the AI area [10]. We choose this dataset, because Python is widely used in the AI area and Python projects released in such a reputable AI conference should be representative and high quality. Note that, not all the papers release their code or implement the technique using Python. Thus, we manually inspected the papers and their links to the implementations. Eventually, we obtained 26 Python projects which contain DC issues.

Since **RQ3** aims to evaluate the performance of SMARTPIP in resolving the DC issues of *Pattern-A*, we apply SMARTPIP , PIP with the backtracking strategy, and Conda to install the dependency libraries for each single Python project in the three datasets. We select PIP with the backtracking strategy and CONDA as the baseline approaches, because they are the representatives of the two categories of dependency resolving techniques as discussed in Section 2.1. We do not include PIP with the legacy strategy in the evaluation, since it will directly lead to build failures.

RQ4 aims to evaluate the performance in resolving the DC issues of *Pattern-B*, We apply SMARTPIP and PIP with the Python built-in virtual environment (i.e.,VENV) to install all the Python projects of each dataset. Note that, since the legacy strategy and the backtracking strategy behaves the same in our empirical study, we adopt PIP with the backtracking strategy. Moreover, as discussed in Section 2.2, PIP VENV, PIPENV, and VIRTUALENV work in a similar mechanism, so we used PIP VENV as the representative technique. We exclude CONDA as the baseline approach in RQ4. This is because, the file size of those library versions downloaded by CONDA is typically much smaller than the one of the same library versions hosted in PYPI. This difference is due to that the library versions in CONDA have been pre-compiled [21]. For example, *numpy-1.21.5* downloaded by CONDA takes up 4.8MB, while the one in PYPI takes up to 15.7 MB. We can hardly make a fair comparison with CONDA due to this inconsistency in the file size of the same library version.

To evaluate both of RQ3 and RQ4, we need to resort to the internet to download the Python libraries from the Python ecosystem. The network bandwidth would affect the speed of download. To make the comparison fair, we set the router used in our experiments and bound the network bandwidth as 10Mbps.

5.2 Answer to RQ3

Table 3 shows the success rate of SMARTPIP and the two baseline approaches. CONDA fails to install the libraries for 86.7% of the projects in the three datasets. These failures are not caused by the dependency resolving mechanism of CONDA. Actually, they are due to the fact that the small-scale Python ecosystem of CONDA (as described in Section 2.3) does not contain the desired libraries that these evaluation subjects require. SMARTPIP successfully solves all the DC issues of *Pattern-A*. PIP with the backtracking strategy failed in the Python project *imgsync* in the WATCHMAN dataset.

Table 4: Time Cost Comparison among PIP, CONDA, and SMARTPIP.

Dataset	PIP - backtracking			Conda				SMARTPIP			
	Deps	T_{Solv}	T_{Total}	T_{Solv}	SPU_{Solv}	T_{Total}	SPU_{Total}	T_{Solv}	SPU_{Solv}	T_{Total}	SPU_{Total}
WatchMan	23	105s	738s	266s	0.39X	855s	0.86X	24s	4.38X	634s	1.16X
HG2.9K	29	119s	2,201s	1,306s	0.09X	2,321s	0.95X	26s	4.58X	2,076s	1.06X
AAAI	23	391s	1,648s	2,893s	0.14X	4,023s	0.41X	8s	48.88X	1,252s	1.31X
Sum	75	615s	4,586s	4,465s	0.14X	7,199s	0.64X	58s	10.60X	3,962s	1.16X

Note: The comparison was conducted on the projects that three approaches successfully build.
Deps: the number of additional libraries to download.
T_{Solv}: the time for dependency resolving.
T_{Total}: the total time to install the dependencies including dependency resolving time and download time.
SPU_{Solv}: the speedup of the dependency resolving time by comparing with PIP.
SPU_{Total}: the speedup of the total time by comparing with PIP.

Table 5: Time Cost Comparison between PIP and SMARTPIP.

Dataset	PIP - backtracking			SMARTPIP		
	Deps	T_{Solv}	T_{Total}	T_{Solv}	T_{Total}	*Speedup*
WatchMan	1,131	1,967s	5,235s	63s	3,264s	1.60X
HG2.9K	202	1,084s	13,535s	89s	11,358s	1.19X
AAAI	2,301	9,561s	30,048s	67s	20,597s	1.46X
Sum	3,634	12,612s	48,818s	235s	35,219s	1.39X

Note: The comparison was conducted on the projects that both SMARTPIP and PIP successfully build.
Deps: the number of additional libraries to download.
T_{Solv}: the time cost for dependency resolving.
T_{Total}: the total time to install the dependencies including dependency resolving time and download time.

This failing case was due to the large search space of the backtracking which has been discussed in Section 3.2.1. SMARTPIP can successfully resolve this DC issue and it costs only 40.71 seconds to complete the installation of all the required libraries.

As CONDA and PIP cannot successfully install the required libraries for all the subjects, to fairly compare the time cost of the three approaches, we conduct the comparison on those subjects (16 in total) where the three approaches successfully build as shown in Table 4. To resolve dependency, PIP requires 75 additional libraries to be downloaded, leading to the time cost of 615 seconds. CONDA leverages SAT solving to resolve dependency, and the time cost including SAT expressions construction and solving is 4,465 seconds, which is much higher than PIP. The speedup of CONDA over PIP in dependency resolving ranges from 0.41X to 0.95X. SMARTPIP leverages SMT solving to resolve dependency, and achieves the highest efficiency compared with the other baseline approaches. The time cost of SMARTPIP in dependency resolving, including SMT expressions construction and solving, is 58 seconds in total, and its speedup over PIP ranges from 1.06X to 1.31X. In terms of the total time cost, SMARTPIP achieves the best performance among the three approaches. The speedup of SMARTPIP over PIP is 1.16X. Note that, as discussed in Section 5.1, the file size of the library versions downloaded by CONDA is smaller than the one from PyPI, and thus CONDA actually requires less time to download the desired libraries. Even with such an advantage, CONDA still achieves the highest time cost due to its expensive dependency resolving.

To better understand the efficiency of SMARTPIP and PIP with the backtracking strategy in resolving DC issues of *Pattern-A* , we include more subjects for evaluation. Table 5 shows the comparison of the subjects (119 in total) that both SMARTPIP and PIP successfully build. To resolve dependency, PIP requires 12,612 seconds to download 3,634 additional libraries for verifying version constraints, while SMARTPIP only costs 235 seconds by SMT solving.

Table 6: Evaluation for RQ4.

Dataset	PIP - venv		SMARTPIP			
	T_{Total}	*Disk*	T_{Total}	*Speedup*	*Disk*	*SR_Rate*
WatchMan	5,408s	7,632	2,388s	2.26X	4,995	34.55%
HG2.9K	13,653s	13,537	2,090s	6.53X	4,974	63.26%
AAAI	30,226s	50,997	5,822s	5.19X	10,066	80.26%
Sum	49,287s	72,166	10,300s	4.79X	20,035	72.24%

Disk: the disk usage by the dependent libraries (MB).
T_{Total}: the total time of installing the dependencies in the virtual environments.
SR_Rate: the rate of the hard disk space reduction.

The speedup of SMARTPIP over PIP with the backtracking strategy ranges from 1.19X to 1.60X in the three datasets.

In summary, SMARTPIP achieves the highest success rate and efficiency in resolving DC issues of *Pattern-A*, comparing with both PIP with the backtracking strategy and CONDA.

5.3 Answer to RQ4

Table 6 shows the comparison results between SMARTPIP and PIP with the built-in virtual environment VENV. In terms of the time cost, the speedup of SMARTPIP over PIP with VENV ranges from 2.26X to 6.53X. The above speedup is mainly due to two reasons. The first one is the same as we discussed in Section 5.2. SMARTPIP is more lightweight than PIP with the backtracking strategy in finding compatible versions of the concerned libraries. The second one is that, SMARTPIP shares the compatible common libraries among different virtual environments and thus avoids having to download libraries for multiple times. Take the HG2.9K dataset as an example. As shown in Table 5, the speedup brought by the constraint solving of SMARTPIP is only 1.19X. In Table 6, when sharing the common compatible libraries among different virtual environments, SMARTPIP achieves a speedup of 6.53X comparing to PIP with VENV, which is much higher than the one brought by the constraint solving.

In terms of the hard disk space, the reduction of SMARTPIP over PIP with VENV ranges from 34.55% to 80.26%. This reduction is mainly because SMARTPIP can share the common compatible libraries among different virtual environments as far as possible. We take Figure 3(b) as an example. To install *tsktsk-0.3.1* and *yamlflow-0.0.9*, PIP isolated the two incompatible libraries *pyyaml* by creating two virtual environments, VENV-1 and VENV-2 respectively. In VENV-1, the project *tsktsk-0.3.1* includes the version constraint *"requests* == 2.24.0*"*, and thus install *requests-2.24.0* eventually. In VENV-2, according to the version constraint *"requests* >= 2.14.2 ∧ *requests*! = 2.18.0*"*, PIP will follow the default policy to download *requests-2.27.1*. Due to the isolation of VENV-1 and VENV-2, two versions of *request* have been downloaded and stored. Our approach

SMARTPIP also creates two virtual environments but shares the libraries in the local repository. After installing *tsktsk-0.3.1* in the first virtual environment, its library *requests-2.24.0* was stored in the local repository. When installing *yamlflow-0.0.9*, SMARTPIP considered that the local library *requests-2.24.0* was compatible with the version constraints *"requests* $>= 2.14.2 \wedge$ *requests*$! = 2.18.0$*"*, and thus would reuse it based on the constraint optimization process.

6 THREATS TO VALIDITY

Internal Validity. The DC issue selection may threaten the validity of our empirical study. To reduce this threat, we leveraged the DC issue dataset collected in the prior study, and manually reproduced them to assure the data quality. Moreover, in the experiment, we limit the network bandwidth to make the comparison fair. However, due to the possibility of the fluctuation of the network bandwidth, the measurement of the time cost for downloading the libraries would likely be inaccurate.

External Validity. The evaluation shows that our technique is effective in resolving DC issues in Python projects. However, to generalize our technique to other software ecosystems would require the domain knowledge about the dependency management and installation strategies. Moreover, the effectiveness of our technique in resolving DC issues of *Pattern-B* should be evaluated by real developers in the actual users' local environment. Performing a user study is important future work.

7 RELATED WORK

Dependency Resolving. Dependency resolving is the core of package management tools. A recent study [23] summarized the characteristics of the existing dependency resolving mechanisms in different package management tools for different software ecosystems. Due to the diversity of dependency resolving mechanisms, DC issues manifest differently. Various recent studies [33, 41–44] have been proposed to understand and detect the DC issues in different software ecosystems. For example, Maven [11], a package management tool for Java ecosystem, adopts the nearest win strategy to resolve dependency when DC issues occur. Some recent empirical studies have found that, DC issues in Maven do not lead to build failure, but would cause either runtime exception [42] or inconsistent semantic behaviors [44]. Based on that, some researchers have proposed static analysis techniques [42] and dynamic analysis techniques [43, 44] to detect DC issues in Maven. NuGet [18], a package management tool for .Net ecosystem, leverages the version constraints to resolve dependency, while it can tolerate some version constraints to be unsatisfied without causing build failures. Wang et al. [33] studied the manifestation patterns and the fixing strategies of DC issues in NuGet, and proposed an efficient approach NUFIX to adjust the version constraints to fix DC issues. Our work targets a different ecosystem, the Python ecosystem, whose package management tools [15, 19, 20] require all the version constraints to be satisfied. A recent study [41] has discovered the manifestation patterns of Python DC issues under the most popular Python package management tool, PIP, and proposed a detection tool WATCHMAN. Different from WATCHMAN, our work reveals the new characteristics of DC issues under the new dependency resolving strategy which was released after WATCHMAN. Our approach can also resolve DC issues without changing the version constraints. Other package management tools which encode dependency resolving as an SAT problem, such as CONDA [15] and COMPOSER [14], are also relevant to our work, as they can be used to resolve DC issues. In contrast to them, our work encodes dependency resolving into an SMT problem for solving, and can mitigate the performance issues by reducing the time complexity.

Build-Failure Repair. Mukherjee et al. [38] studied the relationships between Python project build reproducibility and third-party dependencies, and proposed a tool named PYDFIX to detect and fix the unreproducibility problem. Macho et al. [35] proposed a technique to fix build failures in Maven projects due to outdated dependencies using three repair strategies, namely: updating version, removing dependencies, and adding repositories. By contrast, we derived the optimal solution within the existing version constraints without changing the project's configuration file, in order to find solutions instantly. Horton et al. [30] developed a technique, DOCKERIZEME, for inferring the dependencies required by code snippets automatically. DOCKERIZEME starts with offline knowledge acquisition of the resources and dependencies for popular Python packages from the PYPI ecosystem. It then builds Docker specifications using a graph-based inference procedure. V2 [31] is an upgraded version of the DOCKERIZEME tool, which can work with multiple environments and inference-dependent versions. It uses code execution information and existing API crash information to recommend candidate configurations. Lou et al. [34] proposed a technique HIREBUILD to fix Gradle build failures by mining the historical fixing records. The aforementioned studies derive the fixes of the build failures based on the historical information and common dependency configurations adopted by open source projects. However, such derived fixes can potentially induce the DC issues. Our approach allows developers to promptly fix the build errors of Python projects during installation, by finding the appropriate solutions satisfying all the version constraints.

8 CONCLUSION

In this work, we conducted an empirical study to revisit the characteristics of DC issues under the PIP with backtracking strategy. We found that the resolving strategy for the DC issues of *Pattern-A* suffers from the efficiency issue, while the one for the DC issues of *Pattern-B* would lead to a waste of time and space. Based on the findings, we propose a tool SMARTPIP to overcome the limitations of the fix strategies, and conduct a systematic evaluation. The evaluation results demonstrate that our technique is promising. In the future, we plan to evaluate the usefulness of our approach by deploying it in the real-world developers' environments. The source code of our tool can be found at https://github.com/smartpip/smartpip.

ACKNOWLEDGMENTS

We thank anonymous reviewers for their insightful comments. This work is supported by the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001), Natural Science Foundation of China (61902329), and Xiamen Youth Innovation Fund (3502Z20206036). Rongxin Wu is the corresponding author.

REFERENCES

[1] 2013. PEP 440 – Version Identification and Dependency Specification. https://peps.python.org/pep-0440/

[2] 2018. Issue 7239 of Conda. https://github.com/conda/conda/issues/7239

[3] 2019. Issue 2 of imgsync. https://github.com/alvarolopez/imgsync/issues/2

[4] 2019. Issue 21 of ltiauthenticator. https://github.com/jupyterhub/ltiauthenticator/issues/21

[5] 2019. Issue 8197 of Conda. https://github.com/conda/conda/issues/8197

[6] 2019. Issue 8810 of Conda. https://github.com/conda/conda/issues/8810

[7] 2020. Issue 9983 of Conda. https://github.com/conda/conda/issues/9983

[8] 2020. pip documentation. https://pip.pypa.io/en/stable/cli/pip_install/

[9] 2020. WatchMan issue reports. http://www.watchman-pypi.com

[10] 2021. AAAI Dataset. https://github.com/MLNLP-World/Top-AI-Conferences-Paper-with-Code/blob/master/AAAI/2021/AAAI2021.md

[11] 2021. Maven. https://mvnrepository.com/repos

[12] 2021. PyPI. https://pypi.org

[13] 2022. backtracking resolver. https://pip.pypa.io/en/latest/topics/dependency-resolution/

[14] 2022. Composer. https://getcomposer.org/

[15] 2022. Conda. https://conda.io/

[16] 2022. conda repository. https://docs.conda.io/projects/conda/en/latest/glossary.html#conda-repository

[17] 2022. Issue 11414 of Conda. https://github.com/conda/conda/issues/11414

[18] 2022. NuGet. https://www.nuget.org

[19] 2022. pip. https://pypi.org/project/pip/

[20] 2022. Poetry. https://python-poetry.org/docs/faq/

[21] 2022. Understanding Conda and Pip. https://www.anaconda.com/blog/understanding-conda-and-pip

[22] Pietro Abate and Roberto Di Cosmo. 2011. Predicting upgrade failures using dependency analysis. In *Workshops Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 145–150. https://doi.org/10.1109/ICDEW.2011.5767626

[23] Pietro Abate, Roberto Di Cosmo, Georgios Gousios, and Stefano Zacchiroli. 2020. Dependency Solving Is Still Hard, but We Are Getting Better at It. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2020, London, ON, Canada, February 18-21, 2020*, Kostas Kontogiannis, Foutse Khomh, Alexander Chatzigeorgiou, Marios-Eleftherios Fokaefs, and Minghui Zhou (Eds.). IEEE, 547–551. https://doi.org/10.1109/SANER48275.2020.9054837

[24] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. 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*. ACM, 109–120. https://doi.org/10.1145/2950290.2950325

[25] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings (Lecture Notes in Computer Science, Vol. 4963)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[26] Alexandre Decan and Tom Mens. 2021. What Do Package Dependencies Tell Us About Semantic Versioning? *IEEE Trans. Software Eng.* 47, 6 (2021), 1226–1240. https://doi.org/10.1109/TSE.2019.2918315

[27] Gang Fan, Chengpeng Wang, Rongxin Wu, Xiao Xiao, Qingkai Shi, and Charles Zhang. 2020. Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, Sarfraz Khurshid and Corina S. Pasareanu (Eds.). ACM, 463–474. https://doi.org/10.1145/3395363.3397388

[28] Joseph Hejderup, Arie van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Andrea Zisman and Sven Apel (Eds.). ACM, 101–104. https://doi.org/10.1145/3183399.3183417

[29] Eric Horton and Chris Parnin. 2018. Gistable: Evaluating the Executability of Python Code Snippets on GitHub. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 217–227. https://doi.org/10.1109/ICSME.2018.00031

[30] Eric Horton and Chris Parnin. 2019. DockerizeMe: automatic inference of environment dependencies for python code snippets. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 328–338. https://doi.org/10.1109/ICSE.2019.00047

[31] Eric Horton and Chris Parnin. 2019. V2: Fast Detection of Configuration Drift in Python. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*. IEEE, 477–488. https://doi.org/10.1109/ASE.2019.00052

[32] 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 *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 86–98. https://doi.org/10.1109/ICSE43902.2021.00021

[33] Zhenming Li, Ying Wang, Zeqi Lin, Shing-Chi Cheung, and Jian-Guang Lou. 2022. Nufix: Escape From NuGet Dependency Maze. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*. ACM, 1545–1557. https://doi.org/10.1145/3510003.3510118

[34] 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, ISSTA 2019, Beijing, China, July 15-19, 2019*, Dongmei Zhang and Anders Møller (Eds.). ACM, 43–54. https://doi.org/10.1145/3293882.3330578

[35] Christian Macho, Shane McIntosh, and Martin Pinzger. 2018. Automatically repairing dependency-related build breakage. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, Rocco Oliveto, Massimiliano Di Penta, and David C. Shepherd (Eds.). IEEE Computer Society, 106–117. https://doi.org/10.1109/SANER.2018.8330201

[36] António Morgado, Federico Heras, Mark H. Liffiton, Jordi Planes, and João Marques-Silva. 2013. Iterative and core-guided MaxSAT solving: A survey and assessment. *Constraints An Int. J.* 18, 4 (2013), 478–534. https://doi.org/10.1007/s10601-013-9146-2

[37] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 439–451. https://doi.org/10.1145/3460319.3464797

[38] Suchita Mukherjee, Abigail Almanza, and Cindy Rubio-González. 2021. Fixing dependency errors for Python build reproducibility. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 439–451. https://doi.org/10.1145/3460319.3464797

[39] César Soto-Valero, Nicolas Harrand, Martin Monperrus, and Benoit Baudry. 2021. A comprehensive study of bloated dependencies in the Maven ecosystem. *Empir. Softw. Eng.* 26, 3 (2021), 45. https://doi.org/10.1007/s10664-020-09914-8

[40] Jiawei Wang, Li Li, and Andreas Zeller. 2021. Restoring Execution Environments of Jupyter Notebooks. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*. IEEE, 1622–1633. https://doi.org/10.1109/ICSE43902.2021.00144

[41] 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 *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 125–135. https://doi.org/10.1145/3377811.3380426

[42] 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 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 319–330. https://doi.org/10.1145/3236024.3236056

[43] 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 *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 572–583. https://doi.org/10.1109/ICSE.2019.00068

[44] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2022. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Trans. Software Eng.* 48, 7 (2022), 2295–2316. https://doi.org/10.1109/TSE.2021.3057767