Toward Better Dependency Management in Python Projects

By

Sadman Jashim Sakib

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfillment of the Requirements for
the Degree of Master of Science in Computer Science
at the University of Windsor

Windsor, Ontario, Canada

2025

Toward Better Dependency Management in Python Projects

By

Sadman Jashim Sakib

Date of final oral defence: May 09, 2025

The document has received final approval by the Master's Committee:

Co-Supervisor: Curtis Bright

Co-Supervisor: Muhammad Asaduzzaman

Program Reader: Jessica Chen

Outside Program Reader: Mohammad Hassanzadeh

DECLARATION OF ORIGINALITY

I hereby certify that I am the sole author of this thesis and that no part of this thesis has been published or submitted for publication.

I certify that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis and have included copies of such copyright clearances to my appendix.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

Modern software development heavily relies on third-party packages to accelerate progress, yet two critical challenges persist: managing dependency conflicts during package installation and addressing the frequent absence or incompleteness of configuration files in Python projects. These issues disrupt workflow efficiency, degrade system stability, and hinder reproducibility. This research aims to solve both problems by introducing two separate tools. First, we introduce SMTpip, a tool leveraging Satisfiability Modulo Theories (SMT) solvers to resolve third-party package dependency conflicts and Python version incompatibilities during package installation, ensuring a reproducible and conflict-free environment for Python projects. SMTpip constructs a comprehensive dependency knowledge graph by analyzing metadata from the Python Package Index (PyPI) and translates client project requirements—such as Python version constraints and library dependency constraints—into SMT expressions to find an optimal, conflict-free installation process. Evaluations using four different datasets from open-source software repositories show that SMTpip achieves significant speedups: 39× faster than pip, 37× faster than Conda, 3.2× faster than smartPip, and 4× faster than PyEGo. Additionally, SMTpip is able to determine when a set of dependency constraints is inconsistent, meaning that the constraints are mutually contradictory and there is no way of meeting them all simultaneously. Second, we introduce an automated approach to generating requirements.txt files for Python projects lacking dependency specifications. Our approach addresses the challenges of identifying packages and their compatible versions through code parsing. When tested on 3,081 notebooks, our proposed generator tool successfully generated requirements.txt files and enabled the execution of 1,230 notebooks, achieving a 39.92% success rate—nearly twice that of pipreqs (20.84%, or 642 notebooks). Failures were primarily due to non-dependency issues, highlighting challenges beyond dependency resolution. By ensuring consistent software environments and automating dependency specification, these tools enhance project reproducibility. The implementation of SMTpip and the generator tool are publicly available to facilitate reproducibility.

DEDICATION

I dedicate this thesis to my father Md Jashim Uddin Bhuiyan, who raised me with care and granted me the freedom to pursue my dreams. His guidance has been a cornerstone of my journey, shaping my path with wisdom and trust.

To my mother Salma Begum, whose boundless love and dedication have always been the backbone of our family. Her selfless support and constant encouragement have been a source of strength that words can scarcely capture.

To my wife Rafia, whose unwavering support and patience have carried me through the many challenges of this work. Her belief in me has been my greatest motivation, and her strength has been the foundation upon which this achievement rests.

Finally, to my entire family, whose unconditional love and encouragement have illuminated my path. Their presence has been a guiding light, inspiring me through every step of this journey.

With heartfelt thanks to them all.

ACKNOWLEDGEMENTS

I am profoundly grateful to my supervisors, Dr. Curtis Bright and Dr. Muhammad Asaduzzaman, for their exceptional guidance, mentorship, and unwavering support throughout my research journey. Their deep expertise, insightful feedback, and encouragement have not only shaped this thesis but also enriched my academic growth in countless ways.

My sincere thanks go to my thesis committee members, Dr. Mohammad Hassanzadeh and Dr. Jessica Chen, for their insightful feedback and dedication to ensuring the quality of this thesis.

I am deeply grateful to my friends and family for their unwavering encouragement and belief in me. Their support has been a constant source of strength during this process.

Finally, I would like to acknowledge the School of Computer Science and all those who have assisted me throughout my research. Your help and support have been crucial, and I am truly grateful for it.

TABLE OF CONTENTS

DECLARATION OF ORIGINALITY III

ABSTRACT IV

DEDICATION V

ACKNOWLEDGEMENTS VI

LIST OF TABLES IX

LIST OF FIGURES X

LIST OF ABBREVIATIONS XII

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Research Problems . . . 3
  - 1.2.1 Dependency Conflicts . . . 4
    - 1.2.1.1 Package Dependency Conflicts . . . 4
    - 1.2.1.2 Python Version Incompatibilities . . . 5
  - 1.2.2 Missing or Incomplete Configuration Files . . . 6
- 1.3 Addressing the Research Problems . . . 8
  - 1.3.1 Capabilities and Limitations of Pip . . . 9
  - 1.3.2 Alternative Package Managers . . . 9
  - 1.3.3 Existing Techniques in the Literature . . . 10
  - 1.3.4 Proposed Solutions: SMTpip and Our Automatic Configuration Generator . . . 11
- 1.4 Contributions of the Thesis . . . 12
- 1.5 Outline of the Thesis . . . 12

2 Background and Related Work 14
- 2.1 Package Management . . . 14
- 2.2 Configuration Files . . . 16
- 2.3 Python Package Installation . . . 17
- 2.4 Dependency Constraints . . . 18
- 2.5 Satisfiability Solving . . . 19
  - 2.5.1 Satisfiability Solvers . . . 20
- 2.6 Related Work . . . 22
  - 2.6.1 Dependency Conflict Resolution . . . 23
  - 2.6.2 Environment Reproducibility and Build-Failure Repair . . . 24
  - 2.6.3 SAT and SMT Solvers in Software Engineering . . . 24

3 Dependency Conflict Resolution **26**
3.1 Introduction . . . 26
3.2 SMTpip: SMT-driven approach . . . 29
3.2.1 Knowledge Graph Construction . . . 30
3.2.2 SMT Encoding of Dependency Resolution . . . 33
3.2.3 Example of Dependency Constraints . . . 36
3.3 Comparison With Baseline . . . 39
3.3.1 Comparison with smartPip . . . 39
3.3.2 Comparison with PyEGo . . . 40
3.4 Evaluation . . . 42
3.4.1 Datasets . . . 42
3.4.2 RQ1: How effective is SMTpip in resolving library dependency conflicts and Python version incompatibilities during installation compared to pip, Conda, smartPip, and PyEGo? . . . 45
3.4.3 RQ2: Is the Technique Efficient Enough for Practical Use? . . . 52
3.5 Threats to Validity . . . 54
3.5.1 External Validity . . . 54
3.5.2 Internal Validity . . . 55
3.6 Conclusion . . . 55

4 Generating Missing Requirements.txt Files **57**
4.1 Introduction . . . 57
4.2 The Problem of Missing Dependency Specifications . . . 58
4.3 Proposed Solution and Methodology . . . 60
4.4 Evaluation and Results . . . 62
4.4.1 Evaluation Procedure . . . 62
4.4.2 Results . . . 63
4.4.3 Failure Analysis . . . 64
4.5 Related Work . . . 65
4.6 Threats to Validity . . . 67
4.7 Conclusion and Future Work . . . 67

5 Conclusion **69**
5.1 Summary of Research Findings and Contributions . . . 69
5.2 Future Research Directions . . . 70

REFERENCES **72**

A Sample SMT Instances **79**
A.1 SMTpip: Boolean Variables . . . 79
A.2 smartPip: Integer Variables with Nested Logical Constraints . . . 80
A.3 PyEGo: Nested Logical Constraints . . . 81

VITA AUCTORIS **82**

LIST OF TABLES

3.3.1 Comparison of dependency resolution tools. 41
3.4.1 Datasets used for evaluation and information about the instances in each dataset. 43
3.4.2 Results of SMTpip, pip, Conda and PyEGo in resolving dependencies for consistent cases using the latest dependency knowledge graph. . . 45
3.4.3 Results of SMTpip and smartPip in resolving dependencies for consistent cases using downgraded dependency knowledge graph. 46
3.4.4 Results of SMTpip, pip, Conda and PyEGo in identifying inconsistency among the inconsistent cases using the latest dependency knowledge graph. 48
3.4.5 Comparison of smartPip and SMTpip in Identifying Inconsistencies Across Datasets using downgraded Kgraph. 48
3.4.6 Comprehensive time cost comparison across tools (pip, Conda, smartPip, PyEGo vs. SMTpip). SC (Success Count) is the number of projects where both the tool and SMTpip successfully resolved dependency conflicts. TC (Time Cost) is the time taken for dependency resolution, shown as "Tool TC — SMTpip TC" with Speedup in parentheses (Tool TC / SMTpip TC). Speedup indicates how much faster SMTpip resolves dependencies compared to the respective tool. 53

4.4.1 Success Rate Comparison: SMTpip vs. pipreqs 64

LIST OF FIGURES

1.2.1 An example of a third-party library dependency conflict between direct and transitive dependencies in the ltiauthenticator project, where jupyterhub 5.3.0 requires oauthlib $\geq$ 3.0, conflicting with the project's requirement of oauthlib = 2.*. 5

1.2.2 An example of installation failure due to Python version incompatibilities. The project requires dagit = 1.1.5, which indirectly depends on universal-pathlib, requiring Python $\geq$ 3.7, while the local environment uses Python 3.6.5. 6

1.2.3 Examples of configuration files used to specify dependencies in Python projects, illustrating different formats for documenting version requirements for packages. 7

2.4.1 An example of the declaration of two types of constraints in library the seaborn. 19

3.1.1 An example of a third party library dependency conflict between direct and transitive dependencies. 27

3.1.2 Backtracking in pip's dependency resolution by evaluating multiple versions of pip-tools to find compatibility with click == 6.6. It initially downloads pip-tools 7.4.1 but backtracks through versions 7.4.0, 7.3.0, and 7.2.0, finally selecting pip-tools 4.4.0 as the compatible version. The downloads for each attempted version are shown during this process. 27

3.1.3 An example of installation failure due to Python version incompatibilities. The project requires TensorFlow 2.10.0, which depends on NumPy $\geq$ 1.20.0. However, NumPy $\geq$ 1.20.0 is incompatible with Python 3.6.5, as it requires Python $\geq$ 3.7.0. This results in an installation failure, as indicated by the error messages in the box. 28

3.2.1 SMTpip architecture 30

3.2.2 Data collection process. 31
3.2.3 The knowledge graph illustrates the version-specific dependencies between libraries A and B. Library A has two versions, v1 and v2, while library B has three versions, v1, v2, and v3. Directed edges represent dependency relationships: A v1 depends on B v1, and A v2 depends on B v3. Each version node is annotated with two properties: its release date and Python version constraints. 31
3.2.4 The full process of updating the dependency knowledge graph. . . . 33
3.4.1 Comparison of time taken to generate SMT expression by SMTpip & smartPip for the Watchman dataset. 50
3.4.2 Comparison of Time Taken To Solve SMT Expression by SMTpip & smartPip for the Watchman dataset. 51

4.1.1 An example of a requirements.txt file from the jupyterhub project. . 58
4.2.1 The PyPI package of PySnooper is missing the “requirements.txt” file, causing installation to fail. 59
4.2.2 The PyPI package of NCBImeta is missing the “requirements.txt” file, causing installation to fail. 59
4.3.1 Requirements.txt generator workflow. 62

LIST OF ABBREVIATIONS

SMT	Satisfiability Modulo Theories
SAT	Boolean Satisfiability
CNF	Conjunctive Normal Form
PyPI	Python Package Index
CCS	Computing Classification System
ACM	Association for Computing Machinery
PEP	Python Enhancement Proposal
API	Application Programming Interface
AMO	At-Most-One
CRAN	Comprehensive R Archive Network
SC	Success Count
TC	Time Cost
RQ	Research Question

CHAPTER 1

Introduction

This chapter provides an overview of the thesis, setting the foundation for the research presented in the subsequent chapters. Section 1.1 outlines the motivation behind this work, emphasizing the critical role of package management in the Python ecosystem and the challenges that arise therein. Section 1.2 elaborates on the two primary research problems addressed in this thesis: dependency conflicts and the absence of complete configuration files in Python projects. Section 1.3 discusses the limitations of existing approaches and introduces the proposed solutions to these problems. The contributions of this thesis are detailed in Section 1.4, highlighting the novel tools, datasets, and methods developed. Finally, Section 1.5 provides an outline of the remaining chapters, guiding the reader through the structure of the thesis.

1.1 Motivation

In modern software development, package management is a cornerstone of efficient and scalable code reuse, enabling developers to leverage existing libraries to accelerate project timelines and enhance functionality. Effective package management streamlines the integration of third-party code, ensures compatibility across dependencies, and maintains reproducible environments. This is critical across diverse programming ecosystems. For instance, in R, the Comprehensive R Archive Network (CRAN)[1] hosts packages like ggplot2[2] for data visualization; in Java, Maven[3] manages libraries like

[1] `https://cran.r-project.org/`
[2] `https://ggplot2.tidyverse.org/`
[3] `https://maven.apache.org/`

Apache Commons[4] for utility functions; and in JavaScript, npm[5] distributes packages like React[6] for building user interfaces. These ecosystems highlight the universal need for robust package management to support diverse application domains, from statistical analysis to web development.

Python, with its versatile ecosystem, exemplifies this paradigm, supported by the Python Package Index (PyPI),[7] which hosts over six million package releases [43].[8] Python's prominence in fields like data science, machine learning, and web development amplifies the need for tools that simplify package installation and dependency management [20, 29]. For example, scientific computing relies on NumPy[9] and SciPy;[10] web development leverages frameworks like Django[11] and Flask;[12] and machine learning is driven by libraries like TensorFlow[13] and PyTorch.[14]

Package managers, such as pip[15] for Python, Maven[16] for Java, or npm[17] for JavaScript, and package management platforms like PyPI, CRAN, or npm registries, automate the process of discovering, installing, and updating packages. These tools resolve dependency chains, fetch compatible versions, and integrate packages into projects with minimal manual intervention. For instance, a Python developer can install requests[18] with a single command (i.e., *pip install requests*), and pip ensures that all required dependencies are installed in compatible versions. Similarly, npm automates the installation of express[19] for Node.js applications. This automation reduces errors, saves time, and enhances project reliability. However, challenges persist

[4] https://commons.apache.org/
[5] https://www.npmjs.com/
[6] https://react.dev/
[7] https://pypi.org/
[8] In this thesis, the terms "package" and "library" are used interchangeably.
[9] https://numpy.org/
[10] https://scipy.org/.
[11] https://www.djangoproject.com/
[12] https://flask.palletsprojects.com/
[13] https://www.tensorflow.org/
[14] https://pytorch.org/
[15] https://pypi.org/project/pip/
[16] https://maven.apache.org/
[17] https://nodejs.org/en/learn/getting-started/an-introduction-to-the-npm-package-manager#introduction-to-npm
[18] https://requests.readthedocs.io/
[19] https://expressjs.com/

in managing a large number of interdependent packages, particularly when packages specify conflicting version requirements or demand specific runtime environments, disrupting workflows and undermining reproducibility.

In Python, documenting dependencies is a critical practice to ensure consistent environments across development, testing, and production. The requirements.txt file is a common method, listing packages and their versions (e.g., numpy==1.21.0). However, other approaches exist, such as setup.py, which defines package metadata and dependencies for distribution, and pyproject.toml, a modern standard introduced by PEP 518[20] that supports flexible configuration for build tools like poetry[21] or flit.[22] While requirements.txt is simple and widely used, it lacks metadata for build processes, unlike setup.py or pyproject.toml, which offer greater extensibility for package authors. Despite these options, many projects—particularly legacy or under-documented ones—lack comprehensive dependency documentation, forcing developers to manually reconstruct environments, a process prone to errors and inefficiencies.

The interplay of dependency conflicts, version incompatibilities, and inconsistent documentation across programming ecosystems, including Python, highlights the need for innovative package management solutions. These challenges motivate the research in this thesis, which aims to develop tools that simplify dependency resolution, automate environment setup, and ensure reproducible Python environments, ultimately enhancing developer productivity and project reliability.

1.2 Research Problems

The Python ecosystem presents several installation-related challenges, primarily stemming from two key issues: (1) third-party package dependency conflicts and Python version incompatibilities, and (2) the frequent absence or incompleteness of configuration files in Python projects. These problems are deeply interconnected, as the ability to resolve dependency conflicts depends heavily on the availability and accuracy

[20] https://peps.python.org/pep-0518/
[21] https://python-poetry.org/
[22] https://flit.pypa.io/

of dependency information, typically provided in configuration files. When these files are missing, incomplete, or outdated, managing and resolving conflicts becomes significantly more difficult, undermining the reproducibility and reliability of software environments. Below, we delve into each of these research problems, providing detailed explanations and illustrative examples to underscore their significance and interdependence.

1.2.1 Dependency Conflicts

Package dependency conflicts and Python version incompatibilities are pervasive issues in the Python ecosystem, often leading to installation failures and significant disruptions in development workflows [50, 47, 26, 52]. These problems arise due to the intricate web of dependencies that modern Python projects rely on, where even a single incompatible version can cascade into a series of conflicts [32, 21, 45]. The effectiveness of resolving these conflicts hinges on having complete and accurate configuration files, as their absence or incompleteness makes identifying the root causes of conflicts a manual and error-prone process, linking this issue to the challenges of missing or incomplete dependency specifications.

1.2.1.1 Package Dependency Conflicts

A common issue in Python projects is the conflict between package dependencies, particularly when multiple versions of a package are acceptable, with each version imposing different dependencies, making it difficult to determine which versions of packages to install in order to meet all dependencies simultaneously. For example, consider the open-source project ltiauthenticator,[23] which declares direct dependencies on jupyterhub[24] and oauthlib,[25] requiring jupyterhub ≥ 0.8 and oauthlib $= 2.*$. During installation, pip selects the latest versions that satisfy these constraints, resulting in jupyterhub 5.3.0 and oauthlib 2.1.0. However, jupyterhub 5.3.0 requires oauthlib ≥ 3.0,

[23] `https://github.com/jupyterhub/ltiauthenticator/issues/21`
[24] `https://pypi.org/project/jupyterhub/`
[25] `https://pypi.org/project/oauthlib/`

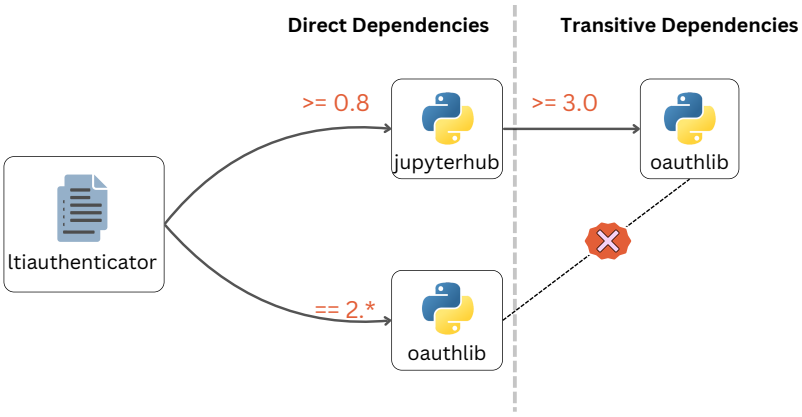


Fig. 1.2.1: An example of a third-party library dependency conflict between direct and transitive dependencies in the ltiauthenticator project, where jupyterhub 5.3.0 requires oauthlib $\geq$ 3.0, conflicting with the project's requirement of oauthlib = 2.*.

which conflicts with the installed oauthlib 2.1.0. This transitive dependency conflict, as illustrated in Figure 1.2.1, exemplifies how such issues can derail the installation process, leading to errors or runtime failures if not carefully managed.

1.2.1.2 Python Version Incompatibilities

Beyond inter-package conflicts, incompatibilities with specific Python versions further complicate the installation process. Figure 1.2.2 illustrates an installation failure under Python 3.6.5 for the dagit[26] package. The user specifies dagit = 1.1.5, which depends on dagster,[27] which in turn requires universal-pathlib.[28] Available versions of universal-pathlib (e.g., 0.0.20, 0.0.21) require Python $\geq$ 3.7, but the local environment runs Python 3.6.5. This mismatch occurs because package dependencies often specify minimum Python version requirements in their metadata, and pip cannot find a compatible version of universal-pathlib for Python 3.6.5, resulting in an installation error. One solution is to create a virtual environment with a compatible Python version, such as Python 3.7, which aligns the local environment with the package's requirements, enabling successful installation. However, this approach requires knowing the exact Python version compatible with all direct and transitive dependencies, which can be

[26] `https://pypi.org/project/dagit/`
[27] `https://pypi.org/project/dagster/`
[28] `https://pypi.org/project/universal-pathlib/`

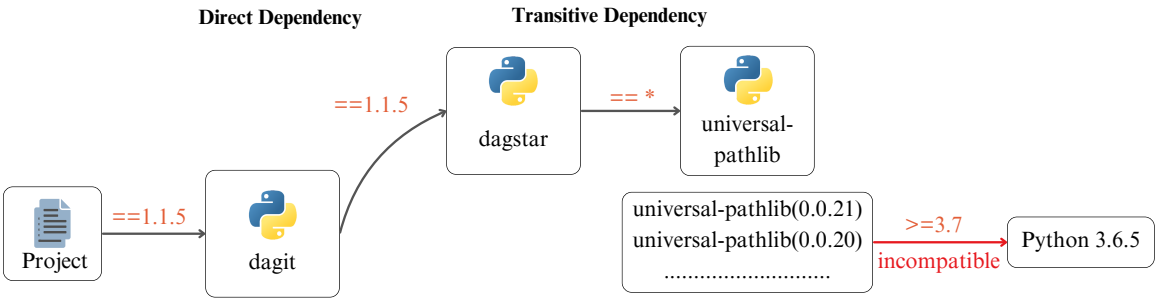


Fig. 1.2.2: An example of installation failure due to Python version incompatibilities. The project requires dagit = 1.1.5, which indirectly depends on universal-pathlib, requiring Python $\geq$ 3.7, while the local environment uses Python 3.6.5.

challenging for large numbers of interdependent packages. This example underscores how the local environment's Python version can conflict with transitive dependency requirements, preventing successful installations even when direct dependencies appear satisfied, and highlights the role of virtual environments in achieving reproducible setups when dependency constraints are well-understood.

Recent studies indicate that resolving these conflicts, whether library-related or Python version-related, requires substantial time and collaboration between upstream and downstream developers, underscoring the severity of the issue [50, 30]. The reliance on configuration files to specify dependencies means that incomplete or missing files exacerbate these challenges, as developers cannot accurately determine the required versions, further complicating conflict resolution.

1.2.2 Missing or Incomplete Configuration Files

The second research problem centers on the frequent absence or incompleteness of configuration files in Python projects. Configuration files, such as requirements.txt, setup.py, or pyproject.toml, are critical artifacts that specify the exact versions of third-party libraries a project depends on, ensuring consistent installations across different environments. However, in many cases—particularly with legacy systems, open-source contributions, or poorly documented projects—these files are either missing, incomplete, or outdated, leading to significant challenges in reproducing the development environment. This issue exacerbates the first research problem, as

setup.py

```
import setuptools

setuptools.setup(
    name="ramanujan",
    version="0.0.1",
    description="Ramanujan Machine",
    packages=['ramanujan'],
    install_requires=[
        'cycler>=0.10.0',
        'kiwisolver>=1.1.0',
        'matplotlib>=3.2.0',
        'mpmath>=1.1.0',
        'numpy>=1.18.1',
        'ordered-set>=3.1.1',
        'pandas>=1.0.1',
        'protobuf>=3.11.3',
        'PyLaTeX>=1.3.1'
    ]
)
```

pyproject.toml

```
[project]
# https://peps.python.org/pep-0621/#readme
requires-python = ">=3.8"
dynamic = ["version"]
name = "pip-tools"
description = " keeps your pinned dependencies fresh."
readme = "README.md"
authors = [{ "name" = "Driessen", "email" = "me@nvie.com" }]
license = { text = "BSD" }
keywords = ["pip", "requirements", "packaging"]
dependencies = [
  # direct dependencies
  "build >= 1.0.0",
  "click >= 8",
  "pip >= 22.2",
]
[build-system]
requires = ["setuptools>=63", "setuptools_scm[toml]>=7"]
build-backend = "setuptools.build_meta"
```

requirements.txt

```
alembic>=1.4
async_generator>=1.9; python_version < '3.10'
certipy>=0.1.2
idna
importlib_metadata>=3.6; python_version < '3.10'
jinja2>=2.11.0
jupyter_events
oauthlib>=3.0
packaging
pamela>=1.1.0; sys_platform != 'win32'
prometheus_client>=0.5.0
psutil>=5.6.5; sys_platform == 'win32'
pydantic>=2
python-dateutil
requests
SQLAlchemy>=1.4.1
tornado>=5.1
traitlets>=4.3.2
```

Fig. 1.2.3: Examples of configuration files used to specify dependencies in Python projects, illustrating different formats for documenting version requirements for packages.

dependency conflict resolution tools rely on accurate dependency specifications to function effectively.

Configuration files serve as a blueprint for the project's dependencies, allowing developers to recreate the exact environment in which the software was developed. This is essential for ensuring that the project functions as intended across various platforms, avoiding discrepancies caused by differing library versions. Without these files, developers must manually deduce the necessary libraries and their versions, a process that is both time-consuming and error-prone. This manual effort often involves trial and error, leading to potential incompatibilities and installation failures when the software is shared or deployed. Examples of such configuration files, including pyproject.toml,[29] setup.py,[30] and requirements.txt,[31] are shown in Figure 1.2.3.

The absence of complete configuration files undermines the reproducibility of Python projects, a cornerstone of reliable software development. For instance, consider a machine learning project that relies on specific versions of libraries like TensorFlow[32] and NumPy.[33] If configuration files are missing, a developer attempting to run the project on a different machine may inadvertently install incompatible versions, leading to runtime errors or divergent behavior. This not only hampers collaboration but also

[29] https://github.com/RamanujanMachine/RamanujanMachine/blob/master/setup.py
[30] https://github.com/jazzband/pip-tools/blob/main/pyproject.toml
[31] https://github.com/jupyterhub/jupyterhub/blob/main/requirements.txt
[32] https://pypi.org/project/tensorflow/
[33] https://pypi.org/project/numpy/

complicates debugging and maintenance, as the root cause of issues becomes harder to trace.

Moreover, the lack of reliable configuration files directly impacts the resolution of dependency conflicts. For example, if a project's configuration files are outdated (e.g., not reflecting a recent library upgrade), the specified versions may no longer be compatible with the project's code, introducing conflicts or installation failures. In the ltiauthenticator example above, incomplete configuration files might omit the transitive dependency on oauthlib ≥ 3.0 required by jupyterhub 5.3.0, leaving developers unaware of the conflict until runtime. This interdependence underscores how the absence or inaccuracy of configuration files amplifies the challenges of managing dependency conflicts.

In large-scale projects or those with extensive dependency trees, manually reconstructing configuration files is impractical. Developers may overlook transitive dependencies or fail to account for version constraints, resulting in incomplete or incorrect specifications. This gap in current practice highlights the need for automated solutions that can infer and generate accurate dependency lists, thereby enhancing the reliability and portability of Python software.

These two research problems—resolving dependency conflicts and generating missing configuration files—are deeply interconnected, as both stem from the complexities of managing dependencies in Python projects. Addressing them requires a holistic approach that not only resolves conflicts but also ensures that dependency specifications are complete and accurate. In the following sections, we present innovative solutions that leverage advanced computational techniques to tackle these challenges, offering a unified framework for efficient and reliable package management.

1.3 Addressing the Research Problems

In the Python ecosystem, package management tools like pip are fundamental for developers seeking to install third-party libraries efficiently. Pip, the standard package installer for Python, simplifies the process by downloading and installing packages

from the Python Package Index (PyPI).[34] It resolves dependency chains, ensuring that required libraries are installed in compatible versions, which reduces the manual effort needed for managing project dependencies. However, pip has notable limitations, particularly when it comes to resolving dependency conflicts efficiently. It is also not able to generate configuration files.

1.3.1 Capabilities and Limitations of Pip

Historically, pip's legacy dependency resolution algorithm[35] frequently failed in such cases, resulting in installation errors. Since version 20.3, pip has adopted a backtracking-based resolver[36] to improve conflict handling. This strategy downloads multiple versions of a package, beginning with the latest version and retrieving its dependency list. It then checks for conflicts and backtracks to previous versions until a compatible version is found. However, pip does not know in advance how many versions it must try or the computational effort required, which can significantly increase resolution times, especially for deeply nested dependencies. Additionally, if a package depends on a specific Python version which is not present in the environment, pip halts without offering a solution, forcing developers to intervene manually.

Another key limitation is that pip cannot generate or update configuration files like requirements.txt, setup.py, or pyproject.toml. These files are vital for documenting dependencies and ensuring reproducible environments, yet developers must create and maintain them manually—a process prone to errors, especially in large or dynamic projects.

1.3.2 Alternative Package Managers

Other tools, such as Conda,[37] provide alternative approaches by managing both Python packages and their dependencies within isolated environments. Conda employs a satisfiability (SAT) solver for dependency resolution, offering more robust conflict

[34] https://pypi.org/
[35] https://debuglab.net/2024/01/04/python-pip-error-legacy-install-failure/
[36] https://pip.pypa.io/en/stable/topics/dependency-resolution/
[37] https://docs.conda.io/projects/conda/en/stable/

handling than pip's earlier methods. However, its reliance on multiple solver calls can hinder scalability. The dependency resolution in Conda is powered by a SAT solver, this process involves multiple invocations of a SAT solver to satisfy several optimization criteria:[38] minimizing removals of already-installed packages, maximizing versions of explicitly requested packages, minimizing optional package installs, respecting channel priorities, prioritizing packages with fewer "track_features," and using timestamps as a tiebreaker when priorities are otherwise equal. Each iteration, or refinement, adjusts constraints to better align with these goals or resolve conflicts, requiring the solver to restart. Like pip, Conda also does not generate configuration files, leaving this aspect of package management unresolved.

1.3.3 Existing Techniques in the Literature

Existing tools for resolving Python package dependency conflicts and version incompatibilities can be categorized into three distinct approaches: analyzing source code, examining configuration files, and reviewing runtime error logs. While prior studies [12, 46] have explored installation incompatibilities, these techniques often lack the comprehensiveness needed for the Python ecosystem.

1. Source Code Analysis: Tools like PyEGo [52] and SnifferDog [49] analyze source code to infer dependencies by parsing import statements and project metadata. This method identifies direct dependencies and Python version dependencies.

2. Configuration File Examination: Approaches such as pip [41], smartPip [47] and DockerizeMe [26] parse configuration files (e.g., requirements.txt, setup.py) to extract dependency details. However, their effectiveness depends on the availability of accurate and complete configuration files, which are frequently absent or outdated in practice.

3. Runtime Error Log Review: Tools like PyDFix [37] and ReadPyE [10] examine runtime error logs to diagnose installation failures and propose fixes. While this

[38] `https://www.anaconda.com/blog/understanding-and-improving-condas-performance`

can aid debugging, it is a reactive strategy that requires errors to occur first, rather than preventing conflicts proactively.

These tools typically rely on heuristics or rule-based methods, which are inadequate for handling intricate conflicts involving nested dependencies or Python version incompatibilities. Moreover, they do not address the generation of configuration files, leaving a significant gap in ensuring reproducibility.

1.3.4 Proposed Solutions: SMTpip and Our Automatic Configuration Generator

To tackle the first research problem—package dependency conflicts and Python version incompatibilities—this thesis introduces SMTpip, a package installation program based on a "Satisfiability Modulo Theories" (SMT) encoding. SMTpip constructs a dependency knowledge graph from PyPI metadata, source code, and configuration files, integrating it with a constraint-solving engine. This engine translates dependencies, version constraints, and user requirements into SMT expressions, which an SMT solver resolves to create a conflict-free installation environment. Unlike heuristic-based tools, SMTpip efficiently manages complex dependency graphs and ensures compatibility across all constraints.

For the second problem—generating missing requirements.txt files—we propose a generator tool that automates this process. Leveraging the same dependency knowledge graph as SMTpip, the tool parses import statements from source code, collects project metadata (e.g., release dates), and queries the graph to determine compatible library versions. It produces a candidate requirements.txt file with version ranges, which SMTpip refines to guarantee compatibility and resolve potential conflicts.

Together, SMTpip and our automatic configuration generation tool form a unified framework addressing both research problems. By resolving dependency conflicts and automating configuration file generation, this approach enhances the reliability, reproducibility, and efficiency of Python package management, overcoming the limitations of existing tools and package managers like pip and Conda.

1.4 Contributions of the Thesis

This thesis advances the field of Python package management through the following contributions:

- A public dataset of 1,359 dependency conflicts collected from open-source Jupyter Notebook projects on GitHub,[39] addressing the lack of real-world benchmarks.
- A novel SMT-driven approach for resolving dependency conflicts, combining a dependency knowledge graph with logical constraint solving to ensure conflict-free installations.
- A tool, called SMTpip, implementing the proposed approach to provide an efficient, automated solution for package management.
- An extensive empirical evaluation of SMTpip against state-of-the-art techniques for dependency conflict resolution using four different datasets.
- An automated approach for generating missing configuration files leveraging the dependency knowledge graph and SMTpip to produce conflict-free dependency specifications.
- An evaluation of the proposed approach for generating missing configuration files against state-of-the-art techniques using diverse projects.

1.5 Outline of the Thesis

This chapter (Chapter 1, Introduction) introduces the research problems associated with Python package installation, specifically third-party package dependency conflicts, Python version incompatibilities, and the challenges of missing or incomplete configuration files. It also provides a brief overview of our research contributions, including the development of SMTpip, a novel tool for dependency resolution, and an automated approach for generating missing requirements.txt files.

[39] `https://github.com/`

Chapter 2 reviews package management in Python and other ecosystems, focusing on `requirements.txt` and tools like `pip` and Conda. It discusses dependency constraints, satisfiability solving, and prior work on dependency resolution using SAT/SMT solvers.

Chapter 3 presents SMTpip, which resolves dependency conflicts in Python projects using an SMT-driven approach with a dependency knowledge graph. It leverages a dataset of 1,359 Jupyter Notebook projects and demonstrates superior performance over `pip`, Conda, smartPip, and PyEGo across four datasets.

Chapter 4 focuses on an automated approach for generating requirements.txt files for Python projects lacking dependency specifications. It addresses the challenge of identifying libraries and their compatible versions by parsing project code, offering a solution to enhance environment reproducibility and streamline package management.

Chapter 5 summarizes the key findings and contributions of the thesis. It includes a comprehensive summary of the research in Section 5.1 and discusses potential future research directions in Section 5.2, outlining opportunities to further advance dependency management in the Python ecosystem.

Appendix A concludes the thesis as the final part, presenting sample SMT instances generated by different tools. Titled Sample SMT Instances, this appendix provides concrete examples of the SMT encodings used for dependency resolution, illustrating the practical application of the techniques discussed in Chapter 3.

CHAPTER 2

Background and Related Work

This chapter provides the foundational knowledge and related work pertinent to this thesis, focusing on Python package installation, its associated constraints, and the broader context of package management across software ecosystems. Section 2.1 introduces package management, detailing package managers and repositories for various ecosystems. Section 2.2 describes configuration files used to specify dependencies across different ecosystems, with a real-world example of a Python requirements.txt file. Section 2.3 discusses the Python package installation process, specifically through tools like pip and Conda. Section 2.4 outlines dependency constraints commonly used in Python package configuration files. Finally, Section 2.5 explains satisfiability solving, a key technique underpinning the solutions proposed in this thesis. Following that, we discuss related works in Section 2.6.

2.1 Package Management

Package management is a critical process in software development involving the acquisition, installation, and maintenance of software libraries and their dependencies. A package is a bundled collection of code, typically including libraries, modules, or frameworks, designed to provide specific functionalities that developers can integrate into their projects. Package management systems streamline this process by automating the retrieval, installation, and updating of packages, ensuring compatibility and resolving dependencies—other packages required for a given package to function correctly. These systems rely on package managers and package repositories, which vary

across programming ecosystems. Below, we outline the primary package managers and repositories for several major ecosystems: Python, Java, JavaScript, and R.

Python: The Python ecosystem uses pip as its default package manager, which retrieves packages from the Python Package Index (PyPI), a repository hosting over 6 million package releases as of 2025 [43]. PyPI contains libraries like numpy, pandas, and tensorflow, catering to diverse applications such as data science and machine learning. An alternative manager, Conda, supports Python and other languages, sourcing packages from the Anaconda repository, which includes over 7,500 packages but is smaller than PyPI [13]. Conda is particularly popular for managing scientific computing environments.

Java: Java developers rely on Maven and Gradle as their primary package managers. Maven uses the Maven Central Repository,[1] a vast collection of Java libraries and frameworks, such as Spring and Hibernate, with standardized metadata for dependency resolution. Gradle, often used for its flexibility in Android development, also accesses Maven Central but supports additional repositories like JCenter. Both managers handle dependencies defined in configuration files (e.g., pom.xml for Maven).

JavaScript: The JavaScript ecosystem employs npm (Node Package Manager) and yarn as package managers, with npm being the default. The npm registry,[2] the largest package repository globally, hosts millions of packages, including frameworks like React and Angular. Yarn, an alternative, offers faster installations and deterministic dependency resolution, accessing the same npm registry. These tools manage dependencies specified in package.json files.

R: In the R ecosystem, CRAN (Comprehensive R Archive Network) serves as both the primary package repository and the default package manager through R's built-in functions like install.packages(). CRAN[3] hosts over 20,000 packages, such as ggplot2 for data visualization and dplyr for data manipulation. The Bioconductor repository complements CRAN for bioinformatics packages, managed similarly via R commands.

Each ecosystem's package manager and repository are tailored to its language's

[1] `https://central.sonatype.com/`
[2] `https://docs.npmjs.com/cli/v8/using-npm/package-spec`
[3] `https://cran.r-project.org/`

conventions and its community's needs, but all share the goal of simplifying dependency management and ensuring reproducible builds. However, challenges like dependency conflicts and version incompatibilities persist across ecosystems, motivating the need for advanced solutions like those proposed in this thesis.

2.2 Configuration Files

Configuration files are essential artifacts in software projects, used to specify dependencies, their versions, and other metadata required for package installation and project execution. These files provide a structured way to declare the libraries a project relies on, ensuring consistency across different environments. The format and purpose of configuration files vary across programming ecosystems, reflecting the unique requirements of each language and its package management system. Below, we describe the primary configuration files used in Python, Java, JavaScript, and R, followed by a real-world example of a Python requirements.txt file.

Python: Python projects commonly use two configuration files: requirements.txt and setup.py. The requirements.txt file lists dependencies with version constraints, enabling reproducible installations via pip. For example, a line like numpy $\geq$ 1.20.0 indicates that the project requires numpy version 1.20.0 or higher. The setup.py file, used for packaging and distribution, defines metadata (e.g., package name, version) and dependencies, often for projects published to PyPI. Modern Python projects may also use pyproject.toml for build configuration, as seen in libraries like seaborn.[4]

Java: In Java, Maven projects use `pom.xml` (Project Object Model) to declare dependencies, repositories, and build settings. This XML file specifies libraries using coordinates, such as `org.springframework:spring-core:5.3.9`, where `org.springframework` is the group ID (identifying the organization or project), `spring-core` is the artifact ID (naming the library), and `5.3.9` is the version. Gradle projects use `build.gradle`, a Groovy- or Kotlin-based file, to define similar information with a more concise syntax.

[4] `https://github.com/mwaskom/seaborn/blob/master/pyproject.toml`

JavaScript: JavaScript projects rely on package.json, a JSON file that lists dependencies, scripts, and metadata. Dependencies are specified under dependencies (for production) or devDependencies (for development), with version ranges like `"react": "^16.8.0"` indicating compatibility with React versions starting from 16.8.0.

R: R projects typically use DESCRIPTION files for package metadata and dependencies, specifying required R packages and their versions (e.g., Imports: ggplot2 ($\geq$ 3.3.0)). For non-package projects, scripts may use renv.lock files with the renv package to capture a snapshot of dependencies, similar to Python's requirements.txt.

Configuration files like these are pivotal for dependency management, but their absence or incorrect specification can lead to installation failures, underscoring the need for automated solutions as explored in this thesis.

2.3 Python Package Installation

The installation of Python packages is facilitated by tools like pip and Conda, each offering distinct approaches to managing dependencies and resolving conflicts. The tool pip, the default package installer for Python, fetches packages and their dependencies directly from the Python Package Index (PyPI). Its installation process follows four main stages: identifying dependencies, resolving them, building wheels,[5] and installing the packages.[6] Prior to version 20.3,[7] pip employed a legacy top-down resolution strategy, which installed dependencies level by level and followed a "latest version" principle. However, this often led to conflicts and installation failures. Since version 20.3, pip has adopted a backtracking-based resolution strategy, iteratively verifying all dependencies and backtracking to alternative versions if conflicts arise. This shift has significantly improved its ability to handle version mismatches.

In contrast, Conda, developed by Anaconda, Inc., is both a package and environment management tool supporting not only Python but also other languages like C,

[5] https://wheel.readthedocs.io/en/stable/

[6] https://pip.pypa.io/en/latest/cli/pip_install/

[7] https://pip.pypa.io/en/latest/user_guide/#changes-to-the-pip-dependency-resolver-in-20-3-2020

C++, and R. Conda retrieves and caches package metadata locally. Conda's dependency resolution uses a SAT solver, invoked multiple times to optimize criteria such as minimizing package removals, maximizing requested package versions, and respecting channel priorities.[8] The solver iteratively refines constraints to optimize these goals and resolve conflicts. This iterative refinement, while thorough, significantly increases runtime, making Conda slower compared to more streamlined alternatives. Moreover, since its Python-specific package collection (more than 7,500 packages)[9] is smaller than PyPI's, Anaconda recommends combining Conda and pip for package management.[10]

2.4 Dependency Constraints

We describe the constraints commonly used in the configuration files of Python packages. An example of the declarations for these constraints in the library seaborn version 0.13.2 is shown in Figure 2.4.1.

Package version constraints specify the version requirements for third-party dependencies of a client package release. When installing the package, pip resolves and installs dependencies satisfying these constraints. The version constraints [40] are defined using comparison operators:

- `~=` : Specifies a compatible version range, with the meaning that `~=V.N` is equivalent to `>= V.N, == V.*`. The `*` in `== V.*` is a wildcard that matches any patch version for the major version `V`. For example, `== 1.*` accepts versions like `1.0`, `1.1`, `1.2`, etc., as long as the major version is `1`.
- `==` : Version equality.
- `!=` : Version not equal.
- `<=, >=` : Inclusive ordered comparison.
- `<, >` : Exclusive ordered comparison.

[8] https://www.anaconda.com/blog/understanding-and-improving-condas-performance
[9] https://docs.conda.io/projects/conda/en/latest/glossary.html
[10] https://www.anaconda.com/blog/understanding-conda-and-pip

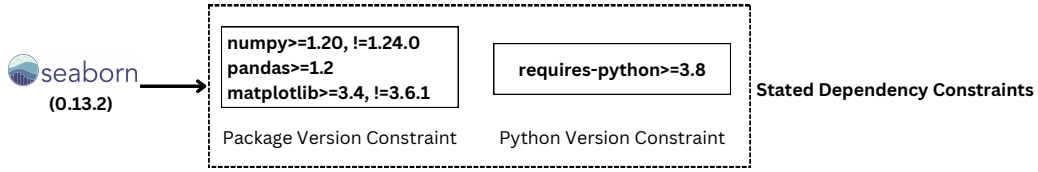


Fig. 2.4.1: An example of the declaration of two types of constraints in library the seaborn.

The Python version constraint defines the compatible Python versions for a package release on PyPI. A package can only be installed if the Python version of the local environment satisfies this constraint. Otherwise, an installation error will occur.

2.5 Satisfiability Solving

The Boolean Satisfiability Problem (SAT) involves determining if there exists a way to assign truth values (true or false) to the variables in a Boolean expression such that the entire expression evaluates to true. The expression is typically given in Conjunctive Normal Form (CNF), which is a conjunction (AND, $\wedge$) of clauses. Each clause is a disjunction (OR, $\vee$) of literals, where literals are either variables (e.g., a) or their negations (e.g., $\neg a$). The notation $a \rightarrow b$ is shorthand for the clause $\neg a \vee b$.

For example, the expression

$$(a \rightarrow \neg b) \wedge (a \rightarrow c) \wedge (\neg c \rightarrow \neg b)$$

consists of the three clauses $(\neg a \vee \neg b)$, $(\neg a \vee c)$, and $(c \vee \neg b)$. Each clause is a disjunction of literals, and the entire expression is a conjunction of those clauses. The goal is to assign truth values to the variables a, b, and c such that the expression evaluates to true, thereby solving the SAT problem. If no truth assignment exists, the problem is said to be *unsatisfiable* or *inconsistent*. SAT solving is an NP-complete problem, and no known algorithms for SAT are efficient in the worst case. Despite this, in practice, a variety of problems can effectively be solved using SAT solvers [8]. Similarly, SMT (Satisfiability Modulo Theories) solvers extend SAT solving by incorporating additional theories, such as arithmetic and bit-vectors, enabling them to handle more

complex constraints.

2.5.1 Satisfiability Solvers

SAT solvers are specialized tools that address the Boolean Satisfiability problem by finding truth value assignments for variables in a Boolean formula that is typically expressed in Conjunctive Normal Form (CNF). They employ advanced algorithms like the Davis–Putnam–Logemann–Loveland (DPLL) procedure [16] and conflict-driven clause learning (CDCL) [36] to efficiently navigate the solution space. Modern implementations, such as MapleSAT [31] and CaDiCaL [5], excel in applications like circuit verification and scheduling, despite the NP-complete nature of SAT. These solvers systematically resolve conflicts and learn from inconsistencies, making them highly effective for practical problem instances [6].

SAT modulo theories (SMT) solvers extend SAT solvers by handling formulas combining Boolean logic with constraints from mathematical theories, such as linear arithmetic, bit-vectors, arrays, or strings. For instance, an SMT solver can determine whether a formula like $(x \geq 3) \wedge (x + y = 5) \wedge (y > 0)$ is satisfiable, where x and y are real numbers, by integrating Boolean satisfiability with arithmetic reasoning. Popular SMT solvers, such as Z3 [17] and cvc5 [2], combine a SAT solver for the Boolean structure of the formula with specialized theory solvers for non-Boolean constraints. Additionally, SMT solvers can incorporate optimization objectives, such as minimizing or maximizing a variable (e.g., finding the smallest x that satisfies the formula), making them suitable for tasks requiring optimal solutions [17].

The distinctions between SAT and SMT solvers are significant, particularly in their scope, capabilities, and applications [3]:

- Expressiveness: SAT solvers are restricted to propositional logic, handling only Boolean variables and their combinations. SMT solvers, however, support a wider range of constraints by incorporating domain-specific theories, enabling them to model complex relationships, such as numerical inequalities or data structure operations.

- Optimization Objectives: SAT solvers are designed to determine whether a given formula is satisfiable, producing one of two outcomes: either a valid assignment that satisfies the formula, or a declaration that the formula is unsatisfiable. Their functionality is limited to this binary result—they natively do not support an objective function. In contrast, SMT solvers extend beyond mere satisfiability by supporting optimization objectives. This capability allows them to not only find a satisfying solution but also identify an optimal one, such as selecting the latest compatible package version in dependency resolution scenarios. An example of SMT solvers' optimization capabilities is their support for *soft constraints*. Unlike hard constraints, which must be satisfied for a solution to be valid, soft constraints are optional—goals the solver attempts to meet but can violate if necessary. Each soft constraint is assigned a *weight*, a numerical value reflecting its relative importance. The solver's objective is to maximize the weighted sum of satisfied soft constraints, balancing trade-offs to achieve an optimal outcome. In our specific approach to dependency resolution, we leverage this feature by formulating the problem as a *weighted MaxSMT problem* [7]. Here, hard constraints enforce essential requirements, such as ensuring all dependencies are correctly satisfied, while soft constraints guide the solver toward preferred outcomes—for instance, prioritizing newer package versions or avoiding unnecessary installations. During the solving process, the Z3 solver [17] detects conflicts between the soft and hard constraints. When a set of soft constraints is detected to be in conflict with the hard constraints, Z3 replaces it with a smaller set of soft constraints. This refinement continues until a solution is found that satisfies all hard constraints and the updated set of soft constraints, and this solution will maximize the weighted sum of the satisfied soft constraints.

- Complexity: SAT solvers deal with purely Boolean formulas, which are computationally challenging (NP-complete) but simpler than the mixed Boolean and theory-based constraints of SMT solvers. SMT problems may be undecidable in

certain theories, necessitating advanced heuristics and theory-specific decision procedures.

- Applications: SAT solvers excel in problems reducible to Boolean constraints, such as hardware verification and combinatorial optimization. SMT solvers, with their richer constraint modeling and optimization capabilities, are preferred for tasks like program verification, symbolic execution, and dependency resolution, where constraints are modeled using variable types such as Int, Bool, or Real, and solved within logics like QF_LIA ((Quantifier-Free Linear Integer Arithmetic) or AUFLIA (Arrays, Uninterpreted Functions, Linear Integers with quantifiers).[11]

- Implementation: SAT solvers rely on algorithms like CDCL to find satisfying assignments, while SMT solvers integrate a SAT solver for the Boolean skeleton with theory solvers for non-Boolean constraints, adding complexity but enhancing versatility.

In this thesis, SMT solvers are central to the proposed SMTpip tool, leveraging their ability to encode complex dependency constraints (e.g., version ranges, Python interpreter compatibility) and optimize solutions (e.g., selecting the latest compatible package versions). The distinction between SAT and SMT solvers, particularly the optimization capabilities of SMT solvers, underscores why SMT is well-suited for addressing the dependency resolution challenges explored in this work.

2.6 Related Work

Dependency management in software engineering is complex due to version conflicts and transitive dependencies, which are indirect requirements of a package's dependencies. Traditional package managers like pip, which uses backtracking, systematically evaluates versions to resolve dependencies but require downloading multiple versions of packages in order to find a set of compatible package versions. Conda, relying on the Anaconda repository, faces challenges due to limited package availability compared

[11] `https://smt-lib.org/logics.shtml`

to PyPI. Recent research, described below, addresses these issues through innovative approaches to dependency conflict resolution, environment reproducibility, and build-failure repair.

2.6.1 Dependency Conflict Resolution

Several studies have proposed methods to address dependency conflicts in Python projects. Wang et al. introduced smartPip [47], which resolves Python package dependencies by leveraging global constraint solving to encode package version requirements as SMT expressions. Its contribution lies in resolving dependency conflicts through the use of package metadata. Similarly, Ye et al. [52] and Cheng et al. [11] developed heuristic-driven dependency inference systems using knowledge graphs, which represent dependencies as nodes and edges. These systems infer dependencies from project source code.

Mukherjee et al. [37] and Cao et al. [9] employ reactive strategies, using pip to attempt installations and parsing error messages to resolve dependency conflicts, but these approaches incur high runtime overhead and do not guarantee a compatible solution for all cases due to limitations in error parsing. Tools like DockerizeMe [26] analyzes a Python snippet's code, PyPI package details, and GitHub project setups to identify and include all necessary packages, creating an environment with the latest versions to make the snippet run. V2 [27] enhances this by detecting and fixing errors caused by outdated package versions, using a guided search informed by error messages and a database of past project failures to select compatible versions that restore the snippet's functionality. Other approaches prioritize constraint relaxation. Zhu et al.'s LooCo [48] automatically loosens version constraints to broaden solution spaces, but this risks unstable or insecure configurations. Here, instability refers to potential runtime errors or inconsistent behavior due to untested or incompatible package versions, while insecurity arises from the possible inclusion of outdated versions with known vulnerabilities. Conversely, InstSimulator [54] detects module conflicts but does not address dependency-level incompatibilities.

2.6.2 Environment Reproducibility and Build-Failure Repair

Reproducing executable environments remains a persistent challenge. Horton et al. [25] found that 38% of GitHub Python snippets fail due to dependency issues, highlighting the need for robust dependency management. Tools like Gistable [25] and SnifferDog [49] address post-hoc environment reconstruction: SnifferDog infers dependencies from Jupyter notebooks, while RELANCER [53] updates deprecated APIs. These tools operate reactively, addressing issues after failures occur.

Build failures caused by dependency conflicts have spurred repair-focused tools. Watchman [50] monitors dependency drift and alerts developers, while PyDFix [37] automates fixes for incompatible dependencies. However, these methods often modify project configurations, potentially introducing instability. In the Java ecosystem, Decca [51] and LibHarmo [28] harmonize library versions—selecting compatible versions to ensure consistency—using historical data and API analysis.

2.6.3 SAT and SMT Solvers in Software Engineering

Satisfiability (SAT) solvers, which determine whether a Boolean formula can be satisfied, and Satisfiability Modulo Theories (SMT) solvers, which extend SAT to handle constraints from mathematical theories like linear arithmetic, have become powerful tools in software engineering. Beyond dependency management, SAT solvers are applied in formal verification of hardware and software, model checking to verify system specifications, and test suite minimization to optimize regression testing [6]. For example, Lopez et al. [33] used SAT solvers to minimize test suites, reducing testing overhead while maintaining coverage.

In dependency management, Cox [15] demonstrated that the package version selection problem, often termed "dependency hell," is NP-complete but can be effectively addressed using SAT solvers by modeling version constraints as Boolean satisfiability problems. This approach has been adopted by package managers like Conda [14], which uses SAT solvers to manage dependencies across its extensive ecosystem, and Mamba [34], a faster alternative. Similarly, Nimble [39] employs SAT-based resolution

for multiversioning. Wang et al. [1] applied SAT solvers to dependency harmonization, selecting compatible dependency versions. ReadPyE by Cheng et al. [10] integrates a pre-built knowledge graph with the Z3 SMT solver to analyze error log files, generate constraints, and iteratively optimize solutions until the program executes successfully.

Our work advances these efforts by leveraging SMT solvers, which enable optimization objectives not natively possible with SAT solvers. Unlike SAT-based tools, which are limited to determining satisfiability of Boolean constraints, our SMTpip tool treats dependency resolution as an optimization problem, using weighted max SMT [7] to select the latest compatible package versions. We encode Python's version constraints (e.g., numpy ≥ 1.21, < 2.0) as SMT formulae, leveraging the pySMT library [23], which provides a Python interface to SMT solvers like Z3 [17].

CHAPTER 3

Dependency Conflict Resolution

3.1 Introduction

Sharing and reusing code have become standard practices in contemporary software development. To support the installation and management of third-party packages, modern software ecosystems provide package management tools and a central repository storing third-party packages. Python has gained significant popularity in recent years, driven in part by advancements in machine learning and data analysis [20, 29]. As of now, there have been more than 6 million package releases in the official repository, PyPI (Python Package Index) [43]. However, users face installation problem due to various factors, including third-party package dependency conflicts [50, 47] and Python version incompatibilities [26, 52]. Many software systems depend on third-party packages to reuse their functionalities instead of reinventing everything from scratch. As a result, package installations become more complicated due to the presence of numerous dependencies [32, 21, 45]. Beyond the relationships between packages, conflicts in third-party packages can lead to issues in resolving dependencies between libraries and incompatibilities with specific Python versions [1, 18, 19]. Recent studies show that resolving these dependency conflicts during package installations requires significant time and collaboration between upstream and downstream developers [50, 30].

Figure 3.1.1 shows an example of third party library dependency conflict. In issue #1 from the open-source project fflpy [22], a dependency conflict arises when a client requests library click == 6.6 while also requiring library pip-tools $\geq$ 4.0.0. The

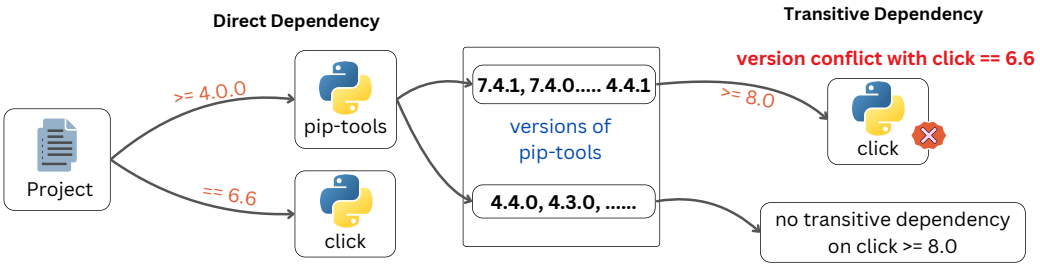
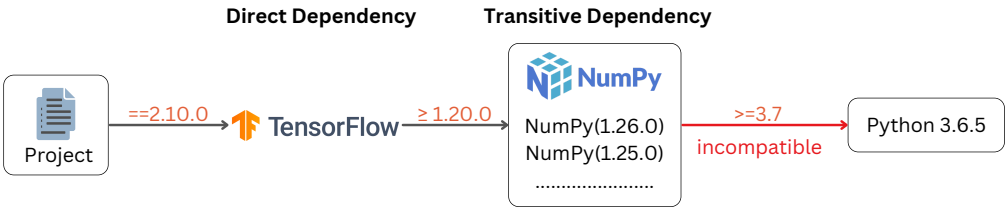


Fig. 3.1.1: An example of a third party library dependency conflict between direct and transitive dependencies.

```
Collecting click==6.6 (from -r .\requirements.txt (line 1))
Using cached click-6.6-py2.py3-none-any.whl (71 kB)

Collecting pip-tools>=4.0.0 (from -r .\requirements.txt (line 2))
Downloading pip_tools-7.4.1-py3-none-any.whl (50 kB)
Downloading pip_tools-7.4.1-py3-none-any.whl.metadata (26 kB)

INFO: pip is looking at multiple versions of pip-tools to determine compatibility.
Attempting other versions:
Downloading pip_tools-7.4.0-py3-none-any.whl (50 kB)
Downloading pip_tools-7.4.0-py3-none-any.whl.metadata (26 kB)
Downloading pip_tools-7.3.0-py3-none-any.whl (45 kB)
Downloading pip_tools-7.2.0-py3-none-any.whl (45 kB)
..........
Downloading pip_tools-4.4.1-py2.py3-none-any.whl.metadata (14 kB)
Downloading pip_tools-4.4.0-py2.py3-none-any.whl (41 kB)

INFO: This process is taking longer than usual. Consider adding stricter constraints.

Installing collected packages: click, pip-tools
Successfully installed click-6.6 pip-tools-4.4.0
```

Fig. 3.1.2: Backtracking in pip's dependency resolution by evaluating multiple versions of pip-tools to find compatibility with click == 6.6. It initially downloads pip-tools 7.4.1 but backtracks through versions 7.4.0, 7.3.0, and 7.2.0, finally selecting pip-tools 4.4.0 as the compatible version. The downloads for each attempted version are shown during this process.




```
INFO: pip is looking at multiple versions of tensorflow to determine which version is compatible with other requirements.
ERROR: Could not find a version that satisfies the requirement numpy>=1.20.0 (from tensorflow==2.10.0) (from versions: none)
ERROR: No matching distribution found for numpy>=1.20.0.
```

Fig. 3.1.3: An example of installation failure due to Python version incompatibilities. The project requires TensorFlow 2.10.0, which depends on NumPy $\geq$ 1.20.0. However, NumPy $\geq$ 1.20.0 is incompatible with Python 3.6.5, as it requires Python $\geq$ 3.7.0. This results in an installation failure, as indicated by the error messages in the box.

conflict occurs because the latest version of pip-tools (7.4.1) has a direct dependency on click $\geq$ 8.0. This introduces a dependency conflict with the project's explicit requirement on click == 6.6. In this scenario, pip's backtracking strategy [41] starts from the latest version of pip-tools (7.4.1) as shown in Figure 3.1.2. It downloads the metadata and checks for conflicts, and after finding the conflict with click == 6.6, pip backtracks to the previous version, 7.4.0. It repeats the process until a non-conflicting version is found (in this case 4.4.0). However, pip does not know how many versions it will need to try or how much computation is required.

Additionally, Figure 3.1.3 illustrates a Python installation failure caused by version constraints. Consider a user attempting to install TensorFlow version 2.10.0 under Python 3.6.5. According to its dependency specifications, TensorFlow == 2.10.0 requires numpy $\geq$ 1.20.0. However, all versions of numpy $\geq$ 1.20.0 mandate Python version $\geq$ 3.7.0, as specified in its configuration files.[1] Consequently, pip reports an installation error due to this unresolved dependency chain. This error is particularly opaque to users because neither the initial installation command (pip install tensorflow == 2.10.0) nor pip's error message explicitly surface the root cause: the indirect Python version requirement imposed by numpy. To diagnose the issue, users must manually trace dependency chains or consult external documentation (e.g., GitHub repositories), significantly increasing troubleshooting effort. Although some studies have attempted to address these installation incompatibilities [12, 46], limited

[1] `https://github.com/numpy/numpy/blob/main/pyproject.toml`

research has been conducted to understand and analyze these issues within the Python ecosystem. Existing techniques [47, 52, 26] focus mainly on parsing third-party dependencies during installations, but fail to account for the impact of local settings and user-specific requirements. Most of them apply heuristics or rule-based approaches to resolve dependency conflicts. As a result, these techniques cannot help users resolve installation errors effectively.

Existing tools for Python dependency resolution, such as PyEGo [52] and smartPip [47], face limitations. PyEGo, which infers dependencies via source code analysis, generates SMT expressions with a non-CNF (Conjunctive Normal Form, a conjunction of disjunctions) structure and multiple satisfiability checks, increasing solving overhead. smartPip, which processes configuration files to generate SMT expressions, incurs high computational costs in its SMT expression generation phase, also its non-CNF encoding structure adds significant solver overhead.

SMTpip addresses these shortcomings by formulating dependency resolution as an optimization problem using SMT solvers. It employs a dependency knowledge graph, derived from PyPI metadata, source code, and configuration files, to model package relationships, and encodes version constraints (e.g., numpy $\geq 1.21, < 2.0$) as SMT expressions via the pySMT library [23] with Z3 [17]. By solving weighted max SMT problems [7], SMTpip prioritizes the latest compatible package versions. Evaluated on four datasets—Watchman [50], HG2.9K [25], SD [52], and a new dataset of 1,359 Jupyter Notebook projects from GitHub—SMTpip successfully resolves all consistent dependency cases (where compatible package versions exist), and identifies all inconsistent cases (cases with no solutions). It achieves significant performance gains, running 39 times faster than pip, 37 times faster than Conda, 3.2 times faster than smartPip, and 4 times faster than PyEGo.

3.2 SMTpip: SMT-driven approach

Figure 3.2.1 shows the overview of our proposed approach, SMTpip. The approach consists of two parts: knowledge graph construction and dependency resolution.

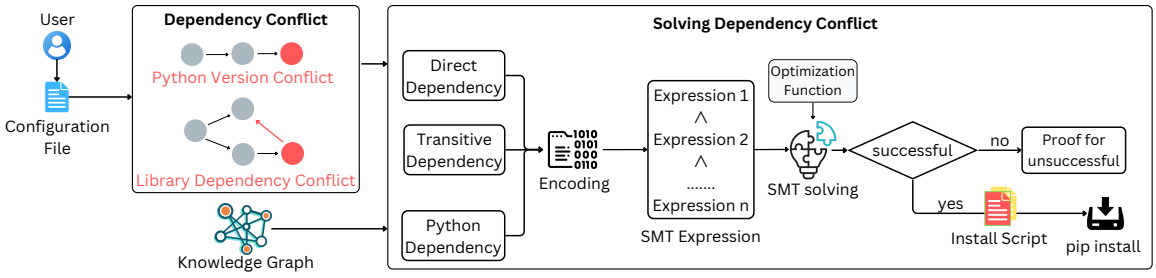


Fig. 3.2.1: SMTpip architecture

This section focuses primarily on constructing the dependency knowledge graph by extracting Python version constraints and dependency version constraints from a central repository of package information. Additionally, it details the encoding of these dependency constraints into SMT expressions.

3.2.1 Knowledge Graph Construction

Since pip with the backtracking strategy is required to iteratively download one candidate version of a concerned library when a dependency conflict occurs, it becomes inefficient when the number and size of iteratively downloaded libraries are large, as discussed in Section 2.3. This is why we construct a knowledge graph in advance. This section describes how the dependency data is collected and modeled into a knowledge graph.

Data Collection: Figure 3.2.2 illustrates the data collection procedure. PyPI, as a central repository for third-party libraries in the Python language, is extensively used in most Python project developments. To collect the necessary data, we implemented a program that retrieves the dependency data from PyPI by collecting the configuration files of each library. At the time of data collection, there were around 6 million releases and nearly half a million packages available on PyPI. The entire process of downloading this information took 10 days.

Dependency Analysis: Once all Python libraries were collected from PyPI, we proceeded to analyze the dependency configuration files for each package. This step involved extracting the version constraints for the dependency libraries as well as the compatible Python version. Section 2.4 outlines how libraries declare their dependency

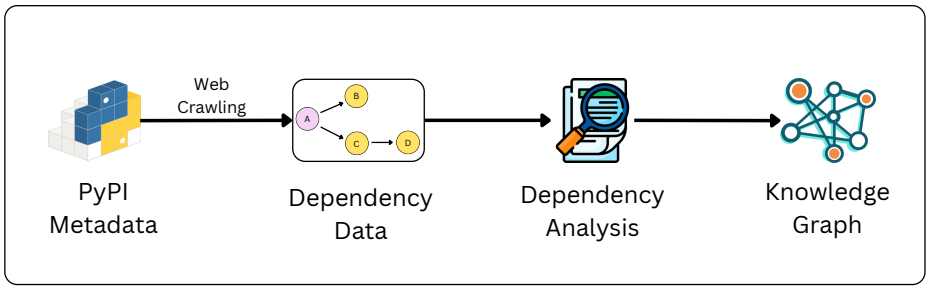


Fig. 3.2.2: Data collection process.

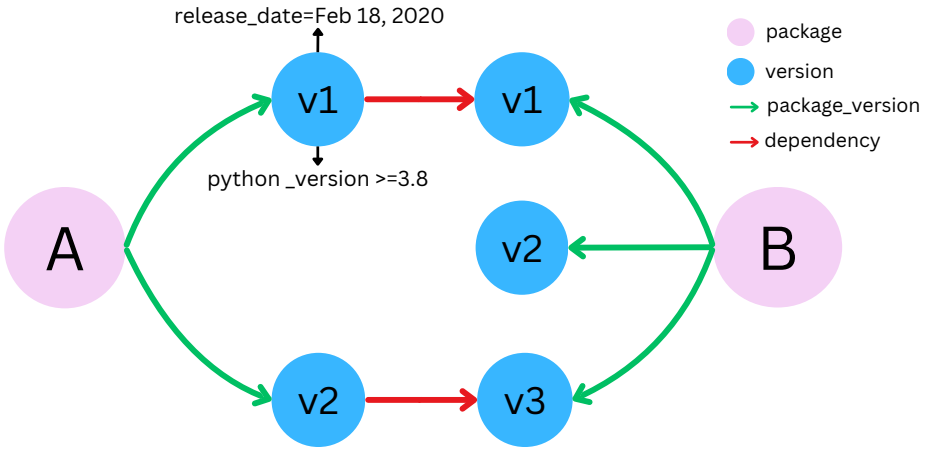


Fig. 3.2.3: The knowledge graph illustrates the version-specific dependencies between libraries A and B. Library A has two versions, v1 and v2, while library B has three versions, v1, v2, and v3. Directed edges represent dependency relationships: A v1 depends on B v1, and A v2 depends on B v3. Each version node is annotated with two properties: its release date and Python version constraints.

constraints in their configuration files.

Knowledge Graph Representation of Package Dependencies: These dependency constraints were modeled into a knowledge graph to represent the relationships between packages, their versions, and respective dependencies, as illustrated in Figure 3.2.3. This approach allows us to visualize and analyze complex dependency structures.

The knowledge graph consists of two types of nodes:

- **Package node:** This node represents individual packages.
- **Version node:** This node represents specific versions of a package. Each version node has two properties: "python_version", which specifies the required Python

version for that particular package version, and "release_date", which indicates the date when the version was released on PyPI.

Additionally, the graph contains two types of edges:

- **Versioning edge:** This edge connects a package node to its respective version nodes, establishing the relationship between a package and the available versions.
- **Dependency edge:** This edge connects the version nodes to the version nodes of the dependent packages, indicating the dependencies between different versions of the packages.

Together, these components form a comprehensive structure that captures the hierarchical relationships and dependencies within the Python package ecosystem. Stored in JSON format with a size of 122 MB, this knowledge graph enables efficient analysis of third-party package dependency conflicts and Python version incompatibilities while also supporting their resolution.

Knowledge Graph Update: Downloading the entire PyPI repository from scratch every time is neither practical nor efficient, as it would be unable to keep up with continuous updates of PyPI. To address this challenge, we implemented an updater program that efficiently retrieves only the necessary changes instead of re-downloading the entire dataset. The updater operates by periodically querying PyPI for updates, identifying modified or newly published packages, and extracting relevant metadata. Once the updates are fetched, it seamlessly integrates them into the existing knowledge graph by modifying dependencies, adding new package entries, and resolving any inconsistencies. To maintain accessibility, the updated knowledge graph is then saved and uploaded back to Google Drive. To further optimize performance, the updater follows a differential update strategy. Daily updates focus on 8,000 popular packages, identified using source rank from libraries.io,[2] ensuring that frequently used libraries stay up-to-date. Meanwhile, a full update of all packages is scheduled weekly to capture broader changes in the ecosystem. Figure 3.2.4 illustrates the diagram of

[2] `https://libraries.io/`

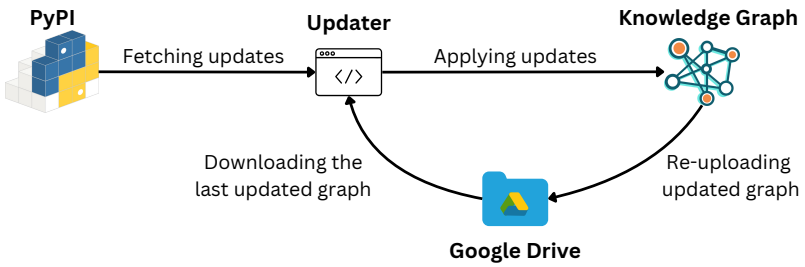


Fig. 3.2.4: The full process of updating the dependency knowledge graph.

this updater process, showcasing the workflow from querying PyPI for updates to integrating the changes into the knowledge graph. This visual representation highlights the steps involved in efficiently updating the knowledge graph without the need for re-downloading the entire dataset.

By leveraging this approach, the updater minimizes redundant data transfers, reduces processing overhead, and ensures that the knowledge graph remains up-to-date with the evolving package landscape of PyPI.

3.2.2 SMT Encoding of Dependency Resolution

Solving the dependency constraints involves addressing two primary types of issues: resolving Python version incompatibilities and resolving library dependency conflicts. These two types of dependency conflicts are the focus of our work, as discussed in Section 1.2.1. In this section, we formalize the process of encoding the dependency resolution problem as a Boolean satisfiability problem.

Given a set of dependency constraints (as described in Section 2.4), our goal is to encode the dependency resolution problem as an instance of the SAT modulo theories (SMT) problem. That is, the resulting SMT instance should have a solution exactly when the dependency resolution problem has a solution. A solution to the SMT instance can also be converted into a solution to the dependency resolution problem.

To do this, for every package p with version v we define a literal p_v representing that version v of package p will be installed. Say that $V(p)$ denotes the set of version numbers of the package p. Since at most one version of any package can be installed

at once, we encode this constraint using the expression

$$\sum_{v \in V(p)} p_v \leq 1.$$

Such expressions are natively represented by a cardinality constraint in our SMT instance. For example, the SMT solver Z3 [17] supports the parameterized theory function "`(_ at-most 1)`" as part of an extension of the SMT-LIB language [4].

Secondly, we need to express the requirements found in the configuration file *requirements.txt*. Each package version constraint in the configuration file determines a subset of the possible versions of package p meeting that requirement. For example, if $V(p) = \{1, 2, 3\}$ the requirement `p >= 2` would only be satisfied by either version 2 or 3 of package p. Suppose C_p is a constraint in the configuration file involving package p and $S(C_p) \subseteq V(p)$ denotes the versions of p meeting the constraint. Then we enforce that some acceptable version of p will be installed using the clause

$$\bigvee_{v \in S(C_p)} p_v.$$

Note that the equality constraint `p == 1` then results in a *unit clause* (a clause of length 1), i.e., just the literal p_1.

Furthermore, installing a package may require other packages to be installed—these are known as *transitive dependencies* and we need to ensure that all transitive dependencies are met on all packages that are installed. Say that the configuration file of version v of package p requires the constraint C_q on package q. That is, if package p version v is installed, then a version package of q whose version number is in the set $S(C_q)$ must also be installed. We represent package dependencies using the clause

$$p_v \rightarrow \bigvee_{u \in S(C_q)} q_u$$

for each package p in the original set of requirements, or in the requirements of any package that p depends on, and so on inductively, until all packages that the original

set may transitively depend on are identified. These packages are determined by computing the components of the dependency knowledge graph containing the vertices corresponding to the original requirements. The graph components are computed using a recursive algorithm with a dictionary to track visited nodes. Starting from the direct dependencies, the algorithm recursively visits all connected transitive dependencies, adding each node to the dictionary once explored. If a node is encountered that is already in the dictionary, the algorithm skips further exploration of that node, ensuring efficient identification of all relevant packages without redundant visits. The Python version constraints are also encoded similarly.

Finally, we would like the solver to prioritize the most recent versions of packages when possible. In order to do this, we define an objective function that the solver maximizes. This function assigns higher values to solutions that select more recent versions and encourages not installing packages at all when possible. Let P denote the set of all packages in the instance. Precisely, the objective function is given by

$$\sum_{p \in P} \left(\sum_{v \in V(p)} \frac{\text{rank}(p_v)}{|V(p)|} \cdot p_v + p_{\text{none}} \right).$$

In this expression, note that the Boolean p_v takes the value 1 if version v of package p is installed (and 0 otherwise), while the Boolean p_{none} takes the value 1 if no version of package p is installed (and 0 otherwise). Also, $\text{rank}(p_v)$ is the rank of version v of package p, defined such that the oldest version has rank 0, the second oldest has rank 1, and so forth, with the most recent version having rank $|V(p)| - 1$.

To ensure consistency, we enforce that at most one of $\{ p_v \mid v \in V(p) \cup \{\text{none}\} \}$ is true. This is encoded in the SMT solver using a cardinality constraint and logical implications. Specifically, we use Z3's `(_ at-most 1)` constraint to ensure that the sum of the Boolean variables $\{ p_v \mid v \in V(p) \} \cup \{p_{\text{none}}\}$ is at most 1, i.e., the cardinality constraint from before is modified to be $\sum_{v \in V(p)} p_v + p_{\text{none}} \leq 1$. Additionally, we add the implications $p_{\text{none}} \rightarrow \neg p_v$ for each version $v \in V(p)$. The weights $\frac{\text{rank}(p_v)}{|V(p)|}$ are normalized, ranging from 0 to $\frac{|V(p)|-1}{|V(p)|} < 1$, ensuring that selecting $p_{\text{none}} = 1$ yields the highest possible value (1) for the parenthesized expression when no version of

p is installed. This structure incentivizes the solver to avoid installing a package unless required by the constraints, while prioritizing the most recent versions when installation is necessary.

3.2.3 Example of Dependency Constraints

Consider the open-source project fflpy [22], with the following dependencies:

- *click* == 6.6
- *pip-tools* $\geq$ 4.0.0

We encode these dependencies along with all transitive dependencies and Python version requirements using Boolean variables and clauses. We define:

- c_v: Boolean variable for version v of *click*, or if v = none this denotes no version of *click* will be installed.
- p_v: Boolean variable for version v of *pip-tools*, or if v = none this denotes no version of *pip-tools* will be installed.
- A: Set of all *click* versions (and the symbol none).
- B: Set of all *pip-tools* versions (and the symbol none).
- $T \subseteq B$: Versions of *pip-tools* $\geq$ 4.0.0, e.g., $\{7.4.1, 7.4.0, \ldots, 4.0.0\}$.

Explicit Clauses

1. **Package Requirements:**
   - *click* == 6.6:

     $$c_{6.6}$$

   - *pip-tools* $\geq$ 4.0.0:

     $$\bigvee_{v \in T} p_v \quad \text{(e.g., } p_{7.4.1} \vee p_{7.4.0} \vee \cdots \vee p_{4.0.0}\text{)}$$

2. **Consistency Constraints:** Ensure at most one version per package using cardinality constraints:

$$\sum_{v \in A} c_v \leq 1 \quad \wedge \quad \sum_{v \in B} p_v \leq 1$$

Cardinality constraints are supported by SMT solvers like Z3 via `(_ at-most 1)`. Although not strictly necessary, we also add the constraints $c_{\text{none}} \rightarrow \neg c_v$ for all $v \in A \setminus \{\text{none}\}$ and $p_{\text{none}} \rightarrow \neg p_v$ for all $v \in B \setminus \{\text{none}\}$.

3. **Transitive Dependencies:**

   - *pip-tools* $==$ 7.4.1 requires *click* $\geq$ 8.0:

   $$p_{7.4.1} \rightarrow (c_{8.0} \vee c_{8.1})$$

   - *pip-tools* $==$ 4.4.1 requires *click* $\geq$ 7.0:

   $$p_{4.4.1} \rightarrow (c_{8.0} \vee c_{8.1} \vee c_{7.0} \vee c_{7.1})$$

4. **Python Version Constraints:**

   - *click* $==$ 6.6 requires *python* $\geq$ 2.7:

   $$c_{6.6} \rightarrow (\mathit{python}_{2.7} \vee \mathit{python}_{2.8} \vee \cdots)$$

   - *pip-tools* $==$ 7.4.1 requires *python* $\geq$ 3.9:

   $$p_{7.4.1} \rightarrow (\mathit{python}_{3.9} \vee \mathit{python}_{3.10} \vee \cdots)$$

5. **Optimization:**

   To encode the objective function in SMT, we formulate a MaxSMT problem using the following weighted soft constraints.

- A soft clause asserting p_{none} with weight 1. This corresponds to the case where no version of *pip-tools* is installed.
- For each version $v \in T$, a soft clause asserting p_v with weight $\frac{\text{rank}(p_v)}{|T|}$, where $\text{rank}(p_v)$ is the rank of version v of *pip-tools* (0 for the oldest version, 1 for the second oldest, ..., $|T| - 1$ for the newest).

For example, there are 47 versions of pip-tools in T, from 4.0.0 ($\text{rank}(p_{4.0.0}) = 0$) to 7.4.1 ($\text{rank}(p_{7.4.1}) = 46$). The soft constraint for p_{none} has weight 1, while version-specific clauses have weights such as $\frac{46}{47} \approx 0.978$ for version 7.4.1 and $\frac{0}{47} = 0$ for version 4.0.0.

The solver maximizes the weighted sum of the satisfied soft constraints. Selecting $p_{\text{none}} = 1$ contributes a weight of 1, while installing a version $p_v = 1$ contributes a weight of $\frac{\text{rank}(p_v)}{47} < 1$, so the soft constraints prioritize not installing a package when possible. When installation is required, newer versions with higher ranks maximize the objective function.

This encoding ensures compatibility and resolves conflicts efficiently, suitable for processing by an SMT solver.

Solving and Validation: Once all dependency constraints are encoded into SMT expressions, we utilize the Z3 SMT solver [17] to determine a satisfying assignment. This assignment specifies compatible versions for the required libraries and a Python version that adheres to all constraints, ensuring a conflict-free installation. Sample SMT instances illustrating this encoding are provided in Sample SMT Instances (Appendix A).

After obtaining the solution (satisfying assignment), a validation step is performed using the knowledge graph. In this step, all direct dependencies of the selected versions of the packages in the solution set are cross-verified. This verification ensures that the selected versions did not introduce any conflicting requirements with other selected packages. In cases where no satisfying assignment exists, the solver produces a proof of the inconsistency of the instance.

3.3 Comparison With Baseline

In this section, we compare different dependency resolution tools, highlighting their strengths and limitations. Table 3.3.1 shows a comparative summary of their capabilities.

3.3.1 Comparison with smartPip

In the smartPip paper by Wang et al. [47], the evaluation methodology involved selectively choosing projects from the datasets. For instance, they included 58 projects from the Watchman dataset that were already solvable by pip's backtracking, and 36 projects from the HG2.9K dataset that exhibited dependency conflict issues. Since these projects were pre-filtered to be solvable by pip, it is unsurprising that smartPip also managed to resolve them successfully. In contrast, our evaluation encompassed all projects in the datasets, regardless of their solvability by pip. This broader and more exhaustive evaluation likely contributed to the lower observed performance of smartPip in our results compared to what was reported in its original paper.

The smartPip tool resolves Python third-party package dependencies through a two-phase process that leverages global constraint solving to encode version requirements as Satisfiability Modulo Theories (SMT) expressions. In the first phase, smartPip parses package requirements and their sub-dependencies, generating SMT constraints to represent valid version ranges. For a requirement like `pandas==1.3.3`, which depends on `numpy>=1.17.3`, the tool recursively processes each dependency, assigning unique IDs to versions (e.g., 1.17.3 to 716459) and creating constraints such as (numpy $\geq$ 716459). However, this phase is inefficient because smartPip's recursive depth-first search (DFS) redundantly processes sub-dependencies. For example, when resolving `pandas==1.3.3`, which requires `python-dateutil>=2.7.3`, smartPip evaluates three satisfying versions (2.7.6, 2.7.5, 2.7.4), each with the same sub-dependency `six>=1.5`. smartPip processes `six>=1.5` separately for each version, redundantly calling its internal function (e.g., `judge_version_range`) to retrieve version lists, check compatibility, and generate identical SMT constraints, significantly

increasing expression generation time. This redundancy worsens in complex graphs where sub-dependencies have their own layered dependencies, amplifying the computational overhead. In the second phase, the Z3 solver evaluates this expression to find a satisfiable version assignment or declare unsatisfiability, indicating that the dependencies cannot be resolved.

A difference in smartPip's approach, compared to SMTpip, arises in its handling of version constraints, dependency traversal, and solution optimization, which can lead to resolution failures and performance degradation in some cases. For example, for the `numpy>=1.17.3` dependency of `pandas==1.3.3`, smartPip interprets $\geq 1.17.3$ as $\sim= 1.17.3$, restricting versions to $\geq 1.17.3$ and $< 1.18.0$. We also uncovered a bug in smartPip—due to an unsorted version list, 1.16.5 is selected as the upper bound rather than 1.18.0. This generates the inconsistent SMT constraints (numpy $<$ 716457) and (numpy $\geq$ 716459). In contrast, SMTpip adheres to PEP 440,[3] encoding $\geq 1.17.3$ to include all versions larger than 1.17.3 (including, for example, 2.0). Additionally, smartPip's recursive DFS has redundant processing as described above, whereas SMTpip employs a dictionary to track visited dependencies, ensuring each sub-dependency, like `six>=1.5`, is processed only once. Furthermore, smartPip's SMT formulation lacks an optimization objective to exclude unnecessary packages not required by the selected version of a direct dependency, such as including packages from unselected `python-dateutil` versions (e.g., 2.7.5, 2.7.4) when only 2.7.6 is chosen, resulting in bloated solutions. SMTpip, however, incorporates optimization objectives to prioritize minimal package sets, ensuring only necessary dependencies are included in the solution. These differences—in constraint interpretation, dependency traversal efficiency, encoding format, and solution optimization—explain why SMTpip resolves some cases where smartPip fails and contributes to the observed performance disparity.

3.3.2 Comparison with PyEGo

Ye et al. propose a tool named PyEGo for dependency resolution [52]. Unlike our approach, PyEGo does not consider the configuration files of a project to resolve

[3] `https://peps.python.org/pep-0440/#version-specifiers`

Table 3.3.1: Comparison of dependency resolution tools.

Tool	Library Dependency Conflict	Python Version Incompatibility	Handles Projects With Multiple Files	Uses Config. Files as Input	Technique to Resolve Conflicts	Encodes as SAT Problem	Gen. Missing Conf. File	Identify Consistent & Inconsistent Cases
pip	✓	✓	✓	✓	Backtracking	✗	✗	✓
Conda	✓	✓	✓	✓	SAT Solver	✓	✗	✓
smartPip	✓	✗	✗	✓	SMT Solver	✓	✗	✗
PyEGo	✓	✓	✗	✗	SMT Solver	✓	✗	✗
SMTpip	✓	✓	✓	✓	SMT Solver	✓	✓	✓

dependencies and cannot handle projects containing multiple Python files. Instead, it takes a single Python file as input and infers dependencies from that file alone. This limitation is why the authors used a dataset of GitHub gists, where each gist is a standalone Python script, and evaluated their approach on 4,836 single Jupyter notebooks.

Additionally, PyEGo determines the required Python version by checking the presence of 19 syntax features corresponding to different Python releases. If an input file does not contain any of these predefined features, PyEGo cannot infer the correct Python version. In contrast, our approach extracts all libraries used across multiple files within a project, considers the Python version dependency information for all versions of all dependent libraries, and encodes these constraints into logical expressions using the SMT encoding techniques discussed in Section 3.2.2.

Furthermore, the encoding techniques differ significantly between PyEGo and SMTpip. PyEGo's SMT expressions, as inferred from available examples, use conjunctions of a Disjunctive Normal Form (DNF)-like structure, which is less efficient for SMT solvers due to the cost of converting such expressions to Conjunctive Normal Form (CNF) [35]. Modern SMT solvers, including Z3, are optimized for CNF, as it aligns with efficient SAT solving techniques like DPLL and CDCL. SMTpip leverages this by specifying its encoding in CNF (or in a form easily converted to CNF in the case of the cardinality constraints) to evaluate all constraints simultaneously, ensuring efficient solving. These differences in encoding structure and solving strategy likely contribute to PyEGo's increased solving time compared to SMTpip.

3.4 Evaluation

This section discusses the evaluation procedure and results of our study. We evaluated the effectiveness of SMTpip in resolving dependency conflicts and Python version incompatibilities using popular open-source Python projects. We answer the following two research questions (RQs):

RQ1: How effective is SMTpip in resolving library dependency conflict and Python version incompatibilities during installation compared to pip, Conda, smartPip and PyEGo?

RQ2: Is the technique efficient enough to be used in practice?

3.4.1 Datasets

As listed in Table 3.4.1, three datasets were selected—Watchman [50], HG2.9K [25], and SD [52]—based on their inclusion of dependency conflicts and Python version incompatibilities, gathered from real-world Python projects across diverse domains. These datasets have also been used in prior studies, further validating their relevance. Additionally, we created a new dataset comprising 190 real-world Jupyter Notebook projects collected from GitHub.[4]

One of the key contributions of SMTpip is its ability to detect inconsistent cases, where it is impossible to satisfy all dependency constraints simultaneously. To verify SMTpip's accuracy in detecting inconsistencies, we performed all detected inconsistent cases using pip and confirmed that SMTpip detection matched. Our findings show that there are 45, 1,223, 24, and 14 inconsistent instances in the Watchman, HG2.9K, SD, and Jupyter Notebook datasets, respectively.

We also measured the size of the dependency graph for each case across all datasets. The size of a dependency graph is defined as the total number of releases involved in generation of SMT instance. For example, a project depends on dagit[5] version 1.1.5, which in turn depends on dagster version 1.1.5, and dagster depends

[4] `https://github.com/`

[5] `https://github.com/dagster-io/dagster/blob/master/pyproject.toml`

Table 3.4.1: Datasets used for evaluation and information about the instances in each dataset.

Dataset	**Total Cases**	**Number of Consistent Cases**	**Number of Inconsistent Cases**	**Dependency Graph Size (Min/Max/Median/Average)**
WatchMan	159	114	45	17 / 6911 / 458 / 864
HG2.9K	2891	1668	1223	6 / 5821 / 139 / 403
SD	100	76	24	13 / 4550 / 93 / 1199
Jupyter Notebook	1359	891	468	50 / 3830 / 780 / 1782

on universal-pathlib (any version); then all versions of universal-pathlib, along with dagit version 1.1.5 and dagster version 1.1.5, form the dependency graph. The total number of releases in this graph represents its size. The maximum dependency graph sizes observed in our datasets are 6,911, 5,821, 4,550, and 3,830 for the Watchman, HG2.9K, SD, and Jupyter Notebook datasets, respectively. The following section briefly describes each dataset.

Watchman: Watchman conducted an empirical study on 235 real-world dependency conflict (DC) issues using PyPI snapshots from July to August 2019. The dataset consists of two types of cases: Pattern A, which includes 210 conflicts caused by remote dependency updates, and Pattern B, which includes 24 conflicts affected by the local environment. Since our focus is primarily on Pattern A cases, we selected only these 210 cases. However, after searching for all the associated projects, we found that 51 repositories had been removed by the time of writing this paper. As a result, we finalized a subset of 159 Python projects for our study. After detecting the consistent and inconsistent cases using SMTpip, we found that there were 114 consistent cases and 45 inconsistent cases in the dataset.

HG2.9K: The dataset consists of 2,891 executable code snippets, designed to facilitate reproducible studies in software engineering. These snippets were collected by scraping gist URLs from the GitHub Gist user interface (UI). After analyzing the dataset with SMTpip, we identified 1,668 consistent cases and 1,223 inconsistent cases.

SD: This dataset was created by the authors of PyEGo and consists of 100 real-world Python projects sourced from GitHub. The selected projects are executable, well-documented, and popular, covering diverse domains such as machine learning, development, security, and more. The dataset includes third-party packages, applications, and tutorials, with an average of 58 Python files, 10,821 lines of code, and 270 import statements per project. After analyzing the dataset with SMTpip, we identified 76 consistent cases and 24 inconsistent cases.

Jupyter Notebook: Our dataset derives from the GH Torrent archive [24], initially containing 247,356 Python-based Jupyter Notebook projects. To ensure reproducibility and relevance, we applied the following filtering and validation steps:

- **Initial Filtering:** We queried the GitHub API to identify projects containing *requirements.txt* files, a standard indicator of explicit dependency specifications. This reduced the dataset to 22,340 projects with dependency metadata.

- **Dependency Graph Analysis:** For each project, we calculated the dependency graph size (i.e., the total number of releases involved in generation of SMT instance). We selected 2,587 projects with dependency graphs exceeding 50 nodes, ensuring a representation of large-scale dependency graphs projects.

- **Conflict Identification:** We employed pip to install dependencies for each project and recorded both successful and failed installations. For successful cases, we analyzed the logs to detect instances where pip used backtracking, identifying those as consistent dependency conflicts. For failure cases, if the failure was due to the absence of a valid version satisfying all constraints, we classified them as inconsistent dependency conflicts. This process identified 1,359 projects with dependency conflicts.

  Consistent instances (891 cases): Conflicts confirmed by pip as resolvable through pip's package installation process. Consistent cases arise when an initial dependency conflict can be resolved by choosing compatible versions of the required packages, allowing all dependencies to co-exist without breaching version constraints.

Table 3.4.2: Results of SMTpip, pip, Conda and PyEGo in resolving dependencies for consistent cases using the latest dependency knowledge graph.

Dataset	Tool	Resolved Dependency			Failed in Resolution		
		Num	Min/Max/Median/Average	Total	Num	Min/Max/Median/Average	Total
WatchMan	pip	114/114 (100%)	1.5 s/641.2 s/11.2 s/29.1 s	3317.4 s	0/114	-	-
	Conda	31/114 (27%)	6.26 s/22.07 s/7.82 s/9.62 s	298.22 s	83/114	4.6 s/18.07 s/4.82 s/4.32 s	358 s
	SMTpip	**114/114 (100%)**	**0.01 s/15.7 s/0.15 s/1.99 s**	**226.86 s**	**0/114**	**-**	**-**
HG2.9K	pip	1668/1668 (100%)	2 s/568.5 s/11.22 s/ 13.9 s	23185.2 s	0/1668	-	-
	Conda	1062/1668 (64%)	3.79 s/77.79 s/10.41 s/12.9 s	13699 s	606/1668	2.49 s/30.69 s/7.61 s/6.3 s	3817.8 s
	PyEGo	1334/1668 (80%)	0.68 s/4.68 s/0.95 s/1.1 s	1467.4 s	334/1668	0.88 s/3.68 s/0.65 s/1.05 s	350 s
	SMTpip	**1668/1668 (100%)**	**0.04 s/4.08 s/0.14 s/ 0.26 s**	**433.68 s**	**0/1668**	**-**	**-**
SD	pip	76/76(100%)	1.3 s/645.1 s/10.53 s/27.02 s	2053.5 s	0/76	-	-
	Conda	24/76 (32%)	11.76 s/72.5 s/21.3 s/25.3 s	607.2 s	52/76	2.6 s/25.15 s/8.5 s/5.3 s	275.6 s
	PyEGo	62/76 (82%)	0.55 s/4.32 s/0.92 s/1.02 s	63.24 s	14/76	0.45 s/5.02 s/0.96 s/1.52 s	21.28 s
	SMTpip	**76/76 (100%)**	**0.03 s/9.31 s/0.17 s/0.48 s**	**36.48 s**	**0/76**	**-**	**-**
Jupyter Notebook	pip	891/891 (100%)	1.7 s/773.2 s/12.48 s/17.16 s	15289.5 s	0/891	-	-
	Conda	784/891(88%)	9.6 s/40.46 s/13.7 s/15.06 s	11805 s	107/891	2.5 s/20.6 s/7.74 s/5.7 s	609.9 s
	SMTpip	**891/891 (100%)**	**0.04 s/9.15 s/0.1 s/0.45 s**	**400.95 s**	**0/891**	**-**	**-**

Inconsistent instances (468 cases): Analysis of the installation failure log files revealed 468 instances of inconsistent conflicts, where no possible selection of package versions could satisfy all constraints simultaneously.

- **Final Dataset Composition:** The resulting Notebook Dataset comprises 1,359 real-world Jupyter Notebook projects with validated dependency conflicts, each containing multiple .ipynb files. With a total of 3,081 *.ipynb* files, this dataset addresses the scarcity of benchmarks for dependency conflict research and is publicly available for reproducibility.

3.4.2 RQ1: How effective is SMTpip in resolving library dependency conflicts and Python version incompatibilities during installation compared to pip, Conda, smartPip, and PyEGo?

This section addresses Research Question 1 (RQ1), which evaluates the effectiveness of SMTpip in resolving library dependency conflicts and Python version incompatibilities

Table 3.4.3: Results of SMTpip and smartPip in resolving dependencies for consistent cases using downgraded dependency knowledge graph.

Dataset	Tool	Successfully Identified Inconsistency			Failed to Identify Inconsistency		
		Num	Min/Max/Median/Average	Total	Num	Min/Max/Median/Average	Total
WatchMan	smartPip	104/107 (97%)	0.6 s/358.2 s/1.1 s/ 5 s	520 s	3/107	0.9 s/248.2 s/1.01 s/ 4.5 s	13.5 s
	SMTpip	**107/107 (100%)**	**0.02 s/13.2 s/0.13 s/1.46 s**	**156.22 s**	**0/107**	-	-
HG2.9K	smartPip	1640/1646 (99%)	0.6 s/39.3 s/ 0.7 s/0.9 s	1476 s	6/1646	0.8 s/54.3 s/ 0.8 s/1.01 s	6.06 s
	SMTpip	**1646/1646 (100%)**	**0.03 s/3.18 s/0.11 s/ 0.16 s**	**263.36 s**	**0/1646**	-	-
SD	smartPip	62/66 (93%)	0.009 s/3.3 s/0.01 s/ 0.6 s	37.2 s	4/66	0.04 s/4.13 s/0.03 s/ 0.5 s	2 s
	SMTpip	**66/66 (100%)**	**0.01 s/6.3 s/0.18 s/0.37 s**	**24.42 s**	**0/66**	-	-
Jupyter Notebook	smartPip	689/711 (96%)	0.01 s/18.49 s/0.1 s/0.55 s	378.95 s	22/711	0.03 s/4.44 s/0.13 s/0.64 s	14.08 s
	SMTpip	**711/711 (100%)**	**0.02 s/7.25 s/0.12 s/0.43 s**	**305.73 s**	**0/711**	-	-

during installation, compared to four existing tools: pip, Conda, smartPip, and PyEGo. The evaluation is conducted across four diverse datasets—WatchMan, HG2.9K, SD, and Jupyter Notebook—each representing a range of real-world scenarios with varying complexity. The results are divided into two parts: the first assesses the tools' ability to resolve consistent dependency cases, as presented in Tables 3.4.2 and 3.4.3, and the second examines their effectiveness in identifying inconsistent cases, as detailed in Tables 3.4.4 and 3.4.5. For fairness in comparisons, evaluations with pip, Conda, and PyEGo utilize the latest dependency knowledge graph, ensuring alignment with the current package ecosystem. In contrast, the comparison with smartPip employs a downgraded knowledge graph, replicating smartPip's outdated dependency information (last updated around 2022), to maintain an equitable assessment given smartPip's reliance on an older knowledge base.

In resolving consistent dependency cases, SMTpip demonstrates improved performance in both success rate and computational efficiency, as evidenced in Table 3.4.2 for comparisons with pip, Conda, and PyEGo, and Table 3.4.3 for comparisons with smartPip. Using the latest dependency knowledge graph, SMTpip achieves a 100% resolution rate across all datasets. For instance, in the HG2.9K dataset, SMTpip resolves all 1,668 cases in 433.68 seconds, with a median resolution time of 0.14 seconds per case, as shown in Table 3.4.2. In contrast, pip, while also achieving a 100% resolution rate, requires 23,185.2 seconds, with a median time of 11.22 seconds per case. Pip's resolution time includes the downloading time of each package's metadata

when it backtracks. Pip's inefficiency stems from its exhaustive exploration of possible dependency versions without prior knowledge of necessary iterations, coupled with the repeated downloading of distribution files during backtracking. Conda, which solves only 64% of the HG2.9K cases in 13,699 seconds, is hampered by its dependence on multiple times of SAT solver invocations, for multiple optimization criteria as described in section 2.3. Each refinement necessitates re-solving the dependency graph with updated constraints, leading to multiple solver invocations and significantly prolonged runtimes. PyEGo, evaluated on HG2.9K and SD datasets due to its limitation to single-file Python scripts, resolves 80% of HG2.9K cases in 1,467.4 seconds. Its heuristic-based strategy pre-selects popular, non-conflicting packages based on import statements before applying SMT solving, prematurely eliminating valid dependency combinations and reducing solution completeness. This restricted scope, excluding configuration files and multi-file projects, limits PyEGo's applicability. In the comparison with smartPip using a downgraded knowledge graph, SMTpip maintains its 100% resolution rate, resolving 1,646 HG2.9K cases in 263.36 seconds, while smartPip achieves a 99% success rate in 1,476 seconds, as shown in Table 3.4.3. smartPip's failures arise from an incorrect handling of version specifiers ($\geq$ and $>$ constraints are treated as $\sim=$), and an unsorted version list that leads to erroneous upper-bound assumptions, preventing the identification of valid solutions even when they exist.

In identifying inconsistent dependency cases, SMTpip exhibits exceptional capability, achieving a 100% identification rate across all datasets, as detailed in Tables 3.4.4 and 3.4.5. With the latest dependency knowledge graph, SMTpip identifies all 1,223 inconsistent cases in the HG2.9K dataset in 134.52 seconds, with a median time of 0.02 seconds per case, as reported in Table 3.4.4. Pip also achieves 100% identification but requires 9,050.2 seconds, attributed to its exhaustive backtracking and repeated downloading of distribution files, which, while thorough, is computationally inefficient for large-scale problems. Conda identifies only 70% of HG2.9K cases in 5,350.6 seconds, failing in the remaining 30% primarily due to the unavailability of certain Python packages or their specific versions in the Anaconda repository. Since the Anaconda repository is smaller than PyPI's comprehensive catalog used by SMTpip and pip,

Table 3.4.4: Results of SMTpip, pip, Conda and PyEGo in identifying inconsistency among the inconsistent cases using the latest dependency knowledge graph.

Dataset	Tool	Successfully Identified Inconsistency			Failed to Identify Inconsistency		
		Num	Min/Max/Median/Average	Total	Num	Min/Max/Median/Average	Total
WatchMan	pip	45/45 (100%)	1.4 s/430.0 s/8.5 s/18.5 s	832.5 s	0/45	-	-
	Conda	32/45 (71%)	5.5 s/10.5 s/8.0 s/5.0 s	160.0 s	13/45	6.3 s/21.9 s/7.9 s/6.5 s	84.5 s
	SMTpip	**45/45 (100%)**	**0.01 s/10.5 s/0.14 s/1.5 s**	**67.5 s**	**0/45**	**-**	**-**
HG2.9K	pip	1223/1223 (100%)	2.1 s/381.0 s/4.7 s/7.4 s	9050.2 s	0/1223	-	-
	Conda	863/1223 (70%)	2.8 s/58.1 s/6.4 s/6.2 s	5350.6 s	360/1223	3.9 s/78.5 s/10.5 s/3.5 s	1260.0 s
	PyEGo	545/1223 (45%)	0.7 s/2.7 s/0.96 s/1.15 s	626.75 s	678/1223	0.8 s/4.9 s/0.98 s/1.2 s	813.6 s
	SMTpip	**1223/1223 (100%)**	**0.001 s/3.10 s/0.02 s/0.11 s**	**134.52 s**	**0/1223**	**-**	**-**
SD	pip	24/24 (100%)	1.2 s/30.0 s/5.6 s/17.5 s	420 s	0/24	-	-
	Conda	11/24 (46%)	5.8 s/43.0 s/21.5 s/15.8 s	173.8 s	13/24	11.7 s/72.0 s/21.2 s/11.5 s	149.5 s
	PyEGo	18/24 (75%)	0.56 s/3.3 s/0.93 s/1.05 s	18.9 s	6/24	0.57 s/4.35 s/0.94 s/1.08 s	6.48 s
	SMTpip	**24/24 (100%)**	**0.004 s/8.4 s/0.016 s/0.35 s**	**8.4 s**	**0/24**	**-**	**-**
Jupyter Notebook	pip	468/468 (100%)	1.34 s/80.5 s/7.5 s/7.5 s	3510 s	0/468	-	-
	Conda	156/468 (33%)	4.7 s/21.0 s/13.8 s/5.2 s	811.2 s	312/468	9.5 s/40.0 s/13.6 s/5.0 s	1560.0 s
	SMTpip	**468/468 (100%)**	**0.005 s/14.5 s/0.52 s/0.55 s**	**257.4 s**	**0/468**	**-**	**-**

Table 3.4.5: Comparison of smartPip and SMTpip in Identifying Inconsistencies Across Datasets using downgraded Kgraph.

Dataset	Tool	Successfully Identified Inconsistency		
		Num	Min/Max/Median/Average	Total
WatchMan	smartPip	52/52 (100%)	0.5 s/258.4 s/1.2 s/4.3 s	223.6 s
	SMTpip	**52/52 (100%)**	**0.01 s/4.5 s/0.14 s/1.2 s**	**62.4 s**
HG2.9K	smartPip	1245/1245 (100%)	0.6 s/19.5 s/0.8 s/1.1 s	1369.5 s
	SMTpip	**1245/1245 (100%)**	**0.001 s/1.10 s/0.01 s/0.07 s**	**87.15 s**
SD	smartPip	34/34 (100%)	0.01 s/2.2 s/0.012 s/0.65 s	22.1 s
	SMTpip	**34/34 (100%)**	**0.014 s/2.4 s/0.036 s/0.25 s**	**8.5 s**
Jupyter Notebook	smartPip	648/648 (100%)	0.012 s/3.5 s/0.11 s/0.45 s	291.6 s
	SMTpip	**648/648 (100%)**	**0.05 s/8.5 s/0.42 s/0.21 s**	**136.08 s**

Conda may encounter projects requiring packages it cannot access. In inconsistent cases, where dependency conflicts exist, the absence of a required package prevents Conda from constructing a complete dependency graph. As a result, Conda fails to declare these cases inconsistent, producing errors like `PackagesNotFoundError` instead of accurately identifying the inconsistency. This limitation accounts for Conda's 30% failure rate in the HG2.9K benchmark, distinguishing it from tools like SMTpip and pip that benefit from PyPI's broader package availability. These limitations contrast with SMTpip's efficient use of a single SMT solving pass and access to PyPI's full package set, enabling complete and rapid identification of all inconsistent cases. PyEGo, limited to HG2.9K and SD datasets due to its single-file script constraint, identifies 45% of HG2.9K cases in 626.75 seconds. In the comparison with smartPip using the downgraded knowledge graph, both tools achieve a 100% identification rate for inconsistent cases, as shown in Table 3.4.5.

However, SMTpip completes the identification of 1,245 HG2.9K cases in 87.15 seconds, compared to smartPip's 1,369.5 seconds. smartPip's longer processing times stem from its recursive DFS approach to parsing dependencies and generating SMT constraints, which redundantly traverses nodes (e.g., exploring a shared dependency like D multiple times in paths $A \to B \to C_1 \to D$ and $B \to C_2 \to D$), leading to inefficiency in large dependency graphs. For instance, the slowest SMTpip instance, which took 100 seconds, involved a project with over 500 packages and 2,000 transitive dependencies, highlighting the computational demands of encoding extensive dependency chains. SMTpip's superior speed and reliability in this context further highlight its effectiveness.

Moreover, the graph in Figure 3.4.1 compares the performance of SMTpip and smartPip in generating SMT expressions for 104 projects from the Watchman Dataset (the instances that both tools successfully solved). The projects are sorted by increasing time required for SMT expression generation. The x-axis represents the number of project files, ordered from least to most time-intensive, ranging from 1 to 104, and the y-axis shows the cumulative time taken in seconds, where each project's time includes the sum of all previous projects' times. SMTpip (blue line with circular markers) and

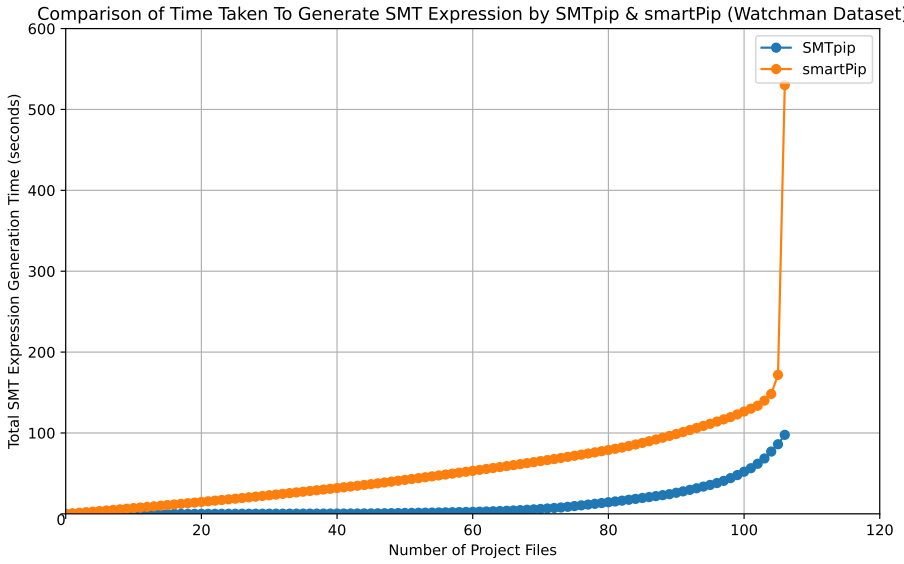


Fig. 3.4.1: Comparison of time taken to generate SMT expression by SMTpip & smartPip for the Watchman dataset.

smartPip (orange line with circular markers) both start with low processing times, but as the projects become more time-intensive, smartPip's cumulative time rises sharply, approaching 500 seconds for the most complex projects, while SMTpip shows a more gradual increase, reaching around 100 seconds.

In conclusion, SMTpip proves highly effective in both resolving consistent dependency cases and identifying inconsistent ones, consistently achieving a 100% success rate across the WatchMan, HG2.9K, SD, and Jupyter Notebook datasets. Its performance surpasses that of pip, Conda, smartPip, and PyEGo, which face distinct challenges: pip's exhaustive backtracking and repeated downloads lead to extended runtimes; Conda's multiple SAT solver invocations for optimization increase computational costs; smartPip's non-CNF encoding structure contributes to resolution delays, as discussed in Section 3.3.1; and PyEGo's non-CNF encoding increase solving overhead, as noted in Section 3.3.2. The use of a downgraded knowledge graph for the smartPip comparison ensures fairness, yet SMTpip still outperforms, establishing itself as a robust and efficient tool for managing library dependency conflicts and Python version incompatibilities during installation.

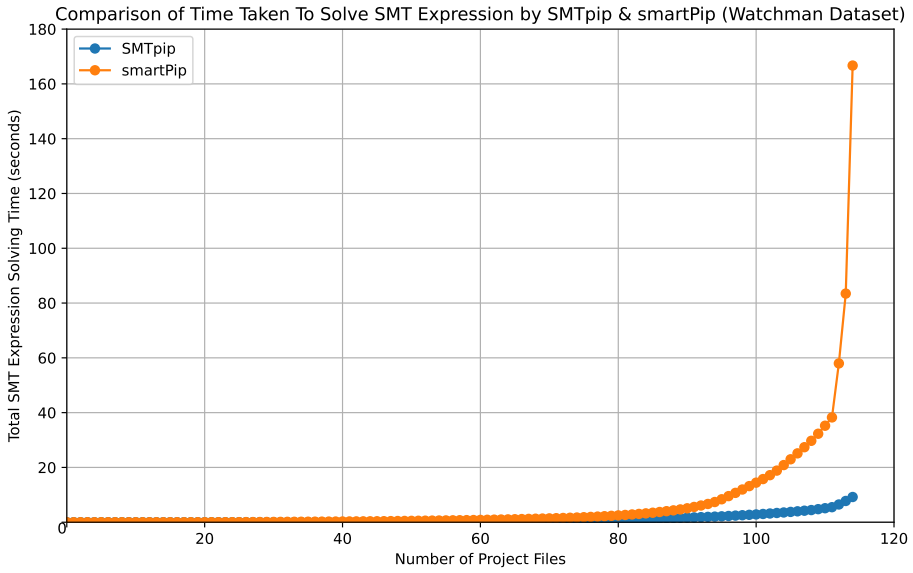


Fig. 3.4.2: Comparison of Time Taken To Solve SMT Expression by SMTpip & smartPip for the Watchman dataset.

RQ1: How effective is SMTpip in resolving dependency conflicts?

In particular:

- SMTpip achieves a 100% success rate in resolving consistent dependency cases across all datasets (WatchMan, HG2.9K, SD, Jupyter Notebook), outperforming pip (100% but slower), Conda (33%–88%), smartPip (93%–99%), and PyEGo (80%–82%).
- For inconsistent cases, SMTpip identifies 100% of conflicts, matching pip's reliability but surpassing Conda (33%–71%), smartPip (100% but less efficient), and PyEGo (45%–75%) in both accuracy and speed.
- SMTpip's effectiveness persists even with a downgraded knowledge graph for fair comparison with smartPip, maintaining 100% resolution and identification rates.
- Challenges faced by other tools stem from pip's exhaustive backtracking, Conda's multiple SAT solver invocations, smartPip's non-CNF encoding structure, and PyEGo's single file scope, non-CNF encoding.

3.4.3 RQ2: Is the Technique Efficient Enough for Practical Use?

The efficiency of SMTpip in resolving library dependency conflicts is critical to determining its suitability for practical use in real-world software development environments. To address Research Question 2 (RQ2), we analyze the speedup achieved by SMTpip compared to established dependency management tools—pip, Conda, smartPip, and PyEGo—across four datasets: WatchMan, HG2.9K, SD, and Jupyter Notebook. Table 3.4.6 presents a comprehensive time cost comparison for projects where both SMTpip and the compared tools successfully resolved dependency conflicts. In this context, Success Count (SC) represents the number of projects with resolved dependency conflicts, Time Cost (TC) denotes the total time required for resolution, and Speedup is calculated as the ratio of the compared tool's time cost to that of SMTpip. The results demonstrate SMTpip's remarkable efficiency, with significant speedups over all tools, underscoring its potential as a practical solution for dependency management.

Table 3.4.6 reveals that SMTpip consistently achieves substantial speedups across all datasets, with overall speedups of 39 times faster than pip, 37 times faster than Conda, 3.2 times faster than smartPip, and 4 times faster than PyEGo for the combined datasets. Against pip, SMTpip's performance is particularly notable: in the HG2.9K dataset, SMTpip resolves 1,668 projects in 433.68 seconds, compared to pip's 23,185.2 seconds, yielding a 53 times faster performance. Pip's prolonged resolution times result from its exhaustive exploration of dependency versions, requiring repeated downloads during backtracking, which increases runtime for large-scale projects. Similarly, Conda exhibits significant delays, with SMTpip being 49 times faster in HG2.9K, where Conda takes 13,699 seconds to resolve 1,062 projects compared to SMTpip's 276.12 seconds. Conda's SAT solver-based resolution, incorporating additional optimization criteria, requires multiple solver invocations, increasing computational overhead.

In the comparison with smartPip, SMTpip achieves a 5.6 times faster performance in HG2.9K, resolving 1,640 projects in 262.4 seconds versus smartPip's 1,476 seconds. smartPip's delays are attributed to its non-CNF encoding structure, which increases

Table 3.4.6: Comprehensive time cost comparison across tools (pip, Conda, smartPip, PyEGo vs. SMTpip). SC (Success Count) is the number of projects where both the tool and SMTpip successfully resolved dependency conflicts. TC (Time Cost) is the time taken for dependency resolution, shown as "Tool TC — SMTpip TC" with Speedup in parentheses (Tool TC / SMTpip TC). Speedup indicates how much faster SMTpip resolves dependencies compared to the respective tool.

Dataset	pip vs. SMTpip		Conda vs. SMTpip		smartPip vs. SMTpip		PyEGo vs. SMTpip	
	SC	TC (pip — SMTpip)	SC	TC (Conda — SMTpip)	SC	TC (smartPip — SMTpip)	SC	TC (PyEGo — SMTpip)
WatchMan	114	3317.4 s — 226.86 s (**14X**)	31	298.22 s — 61.69 s (**5X**)	104	520 s — 151.84 s (**3.4X**)	N/A	N/A
HG2.9K	1668	23185.2 s — 433.68 s (**53X**)	1062	13699 s — 276.12 s (**49X**)	1640	1476 s — 262.4 s (**5.6X**)	1334	1467.4 s — 346.84 s (**4.2X**)
SD	76	2053.5 s — 36.48 s (**56X**)	24	607.2 s — 11.52 s (**52X**)	42	25.2 s — 22.9 s (**1.1X**)	62	63.24 s — 29.76 s (**2.12X**)
Jupyter Notebook	891	15289.5 s — 400.95 s (**38X**)	784	11805 s — 352.8 s (**33X**)	689	378.95 s — 296.27 s (**1.27X**)	N/A	N/A
Sum	**2749**	**43845.6 s — 1097.97 s (39X)**	**1901**	**26409.42 s — 702.13 s (37X)**	**2475**	**2400.15 s — 733.45 s (3.2X)**	**1396**	**1530.64 s — 376.56 s (4X)**

solver overhead, as discussed in Section 3.3.1. PyEGo, evaluated only for HG2.9K and SD due to its design for single-file Python scripts, shows SMTpip being 4.2 times faster in HG2.9K, resolving 1,334 projects in 346.84 seconds compared to PyEGo's 1,467.4 seconds. PyEGo's non-CNF encoding contribute to increased solving times, as noted in Section 3.3.2. In smaller datasets like SD, SMTpip's performance over smartPip is modest (1.1 times faster), reflecting smartPip's efficiency in simpler cases, yet SMTpip still outperforms it. These results, grounded in Table 3.4.6, highlight SMTpip's ability to resolve dependencies significantly faster than its counterparts, often by factors of 3 to 53 times depending on the dataset and tool.

The practical implications of SMTpip's efficiency are evident when considering the cumulative time savings across all datasets. For the 2,749 projects where both SMTpip and pip succeeded, SMTpip completes resolutions in 1,097.97 seconds, compared to pip's 43,845.6 seconds, making it 39 times faster. Against Conda, SMTpip resolves 1,901 projects in 702.13 seconds versus Conda's 26,409.42 seconds, making it 37 times faster. Even against smartPip, which uses a downgraded knowledge graph, SMTpip is 3.2 times faster, resolving 2,475 projects in 733.45 seconds compared to smartPip's 2,400.15 seconds. For PyEGo, SMTpip is 4 times faster across 1,396 projects (376.56 seconds versus 1,530.64 seconds), further underscoring its efficiency. The reasons for these speed improvements tie directly to the inefficiencies of the compared tools: pip's exhaustive backtracking, Conda's multiple SAT solver calls, smartPip's and

PyEGo's use of a non-CNF SMT encoding. In contrast, SMTpip leverages CNF-based Satisfiability Modulo Theories (SMT) instances to efficiently navigate the dependency graph, minimizing redundant computations and ensuring convergence to valid solutions. The consistent and substantial speed improvements across diverse datasets, as detailed in Table 3.4.6, confirm that SMTpip's technique is not only theoretically sound but also highly efficient for practical use, making it a viable and superior alternative to existing dependency management tools in real-world software development scenarios.

RQ2: Is the technique efficient enough to be used in practice?

In particular:

- SMTpip achieves significant speedups over pip (39× overall, up to 56× in SD), driven by pip's inefficient backtracking and repeated downloads.
- Compared to Conda, SMTpip offers a 37× speedup (up to 52× in SD), as Conda's SAT solver requires multiple constraint refinements, inflating runtime.
- Against smartPip, SMTpip provides a 3.2× speedup (up to 5.6× in HG2.9K), overcoming smartPip's non-CNF encoding structure that increases solver overhead.
- SMTpip outperforms PyEGo with a 4× speedup (up to 4.2× in HG2.9K), where PyEGo's non-CNF encoding contribute to increased solving times.

3.5 Threats to Validity

This section discusses threats to the validity of our research. Threats to external validity refer to the generalizability of our findings, and threats to internal validity involve potential biases or errors in our research methodology.

3.5.1 External Validity

Threats to external validity refer to the generalizability of our findings. We evaluate SMTpip using **Python**-based projects. One can argue that the results may not be generalized to every Python project. However, we would like to point to the fact that

we consider four different datasets of varying sizes and covering different application domains. While Watchman, HG2.9K and SD datasets were used by prior studies, we create a new dataset consisting of real-world Jupyter Notebook projects collected from GitHub based on several criteria. Thus, our results should largely carry forward. Other ecosystems (e.g., Java/Maven, JavaScript/npm, R/CRAN) may exhibit different dependency resolution challenges due to variations in versioning schemes, dependency graphs, or package metadata formats. However, the core of our technique does not depend on any specific programming language or ecosystem. Thus, the technique should be applicable to other ecosystems with minor changes.

3.5.2 Internal Validity

Threats to internal validity involve potential biases or errors in our research methodology. SMTpip assumes that package metadata (e.g., *install_requires* in Python) is accurate and complete. In practice, packages may omit dependencies, declare overly permissive version constraints, or rely on implicit environment-specific behavior (e.g., system libraries). Such inaccuracies could propagate errors in the constraint-solving process. The current implementation of SMTpip relies on the Z3 solver for constraint resolution. The configuration of the Z3 solver could potentially affect the results, as it directly influences how constraints are handled and solutions are selected. This aspect was not explored in our experiments. However, the experimental results show that SMTpip is more effective than state-of-the-art approaches despite using the default settings for the Z3 solver. Future work could investigate solver-agnostic formulations or adaptive configuration strategies to enhance the tool's flexibility and performance across different solver configurations.

3.6 Conclusion

Modern Python development is plagued by dependency conflicts and version incompatibilities, which disrupt workflows, compromise reproducibility, and degrade system stability. This thesis introduces SMTpip, a novel SMT-based approach that addresses

these challenges by combining a dynamically constructed dependency knowledge graph with constraint-driven optimization. By translating dependency rules, Python version constraints, and user requirements into SMT expressions, SMTpip generates conflict-free installation plans. It supports both configuration-aware resolution, for projects with dependency files like `requirements.txt`, and code-driven inference, which analyzes source code (e.g., import statements) to infer dependencies when explicit dependency configuration files are absent.

Our evaluation in four datasets, three benchmark datasets (Watchman, HG2.9K, SD), and a newly curated collection of 1359 real-world Jupyter Notebook projects, demonstrate the superiority of SMTpip. It successfully resolved all dependency conflicts and version incompatibilities for all the consistent cases and accurately identified all the inconsistent cases, outperforming state-of-the-art tools like pip, Conda, smartPip, and PyEGo by significant margins in speed (up to 39× faster than pip). These results validate the efficiency of SMT solvers in navigating Python's complex dependency ecosystem, avoiding backtracking and combinatorial explosions inherent in traditional methods.

Key contributions include: (1) a SMT-driven dependency resolution framework that ensures compatibility during installation, (2) a public dataset of real-world dependency conflicts to advance reproducibility research, and (3) empirical validation of SMT's viability in dependency management. By bridging the gap between installation-time conflict resolution and post-hoc environment repair, SMTpip reduces developer effort, enhances system stability, and promotes reproducible software environments.

Future work includes extending SMTpip to other ecosystems (e.g., JavaScript, Java) and integrating incremental resolution strategies for evolving dependencies. The success of SMTpip underscores the potential of formal methods in dependency management, offering a pathway toward more robust, efficient, and scalable software ecosystems.

CHAPTER 4

Generating Missing Requirements.txt Files

This chapter introduces an automated approach to generating a *requirements.txt* file for Python projects lacking dependency specifications, addressing the challenges of identifying packages and their compatible versions through code parsing.

4.1 Introduction

The creation of a *requirements.txt* file is essential for ensuring the reproducibility and portability of Python projects. This file specifies all external packages and their versions required to run a project, allowing developers to recreate the same environment across different systems. For example, a typical *requirements.txt* file for a web-based project "jupyterhub"[1] include packages like `jupyter_events`, `pydantic`, and `SQLAlchemy` with specific version constraints, as shown in Fig. 4.1.1. However, many projects, particularly legacy or poorly documented ones, lack this file,[2] creating significant obstacles[3] for deployment and maintenance. Manually reconstructing a *requirements.txt* file is labor-intensive and prone to errors,[4] as it involves identifying all imported packages and selecting compatible versions.

To address this challenge, we introduce a standalone generator tool that automates

[1] `https://github.com/jupyterhub/jupyterhub/blob/main/requirements.txt`
[2] `https://github.com/ktmeaton/NCBImeta/issues/10`
[3] `https://github.com/cool-RR/PySnooper/issues/26`
[4] `https://stackoverflow.com/questions/31684375/automatically-create-file-requirements-txt`

requirements.txt ⟶

```
alembic>=1.4
certipy>=0.1.2
jinja2>=2.11.0
jupyter_events
oauthlib>=3.0
pydantic>=2
SQLAlchemy>=1.4.1
tornado>=5.1
traitlets>=4.3.2
```

Fig. 4.1.1: An example of a requirements.txt file from the jupyterhub project.

the creation of a *requirements.txt* file by analyzing project files and leveraging historical version data from PyPI, guided by the project's development timeline. This work tackles the second research problem of this thesis: generating a *requirements.txt* file when none exists. It complements the first research problem, addressed by SMTpip (Chapter 3), which uses a Satisfiability Modulo Theories (SMT) encoding to resolve dependency conflicts and Python version incompatibilities for an existing *requirements.txt* file. The generator operates independently, relying solely on code parsing and PyPI version data. Together, these tools form a powerful pipeline: the generator produces a *requirements.txt* file, which SMTpip can then use to ensure a conflict-free environment. This chapter outlines the problem, the generator's methodology, and its role in enhancing the reproducibility of Python projects.

4.2 The Problem of Missing Dependency Specifications

In Python projects, the absence of a *requirements.txt* file creates significant challenges that impede installation, deployment, and maintenance. A primary issue is that automated installation processes, such as those using `pip`, often depend on this file to identify and install the necessary dependencies. When the file is missing, the installation can fail entirely, as demonstrated by real-world examples from the

```
$ pip install pysnooper
Collecting pysnooper
  Using cached https://files.pythonhosted.org/packages/43/58/e36822363b00e3a7f15621b5b34587e171521c5
    Complete output from command python setup.py egg_info:
    Traceback (most recent call last):
      File "<string>", line 1, in <module>
      File "/tmp/pip-install-blixjg7f/pysnooper/setup.py", line 16, in <module>
        install_requires=open('requirements.txt', 'r').read().split('\n'),
    FileNotFoundError: [Errno 2] No such file or directory: 'requirements.txt'

    ----------------------------------------
Command "python setup.py egg_info" failed with error code 1 in /tmp/pip-install-blixjg7f/pysnooper/
```

Fig. 4.2.1: The PyPI package of PySnooper is missing the "requirements.txt" file, causing installation to fail.

```
$pip install NCBImeta

Collecting NCBImeta
  Using cached NCBImeta-0.6.5.tar.gz (26 kB)
    ERROR: Command errored out with exit status 1:
     command: /home/program2/bin/NCBImeta/0.6.5/bin/python3 -c 'import sys, setuptools, tokenize; sys.argv[0] =
         cwd: /tmp/pip-install-5gcxpj4l/NCBImeta/
    Complete output (5 lines):
    Traceback (most recent call last):
      File "<string>", line 1, in <module>
      File "/tmp/pip-install-5gcxpj4l/NCBImeta/setup.py", line 6, in <module>
        with open("requirements.txt", 'r') as r:
    FileNotFoundError: [Errno 2] No such file or directory: 'requirements.txt'
    ----------------------------------------
ERROR: Command errored out with exit status 1: python setup.py egg_info Check the logs for full command output.
```

Fig. 4.2.2: The PyPI package of NCBImeta is missing the "requirements.txt" file, causing installation to fail.

PySnooper[5] project, a debugging tool that traces Python code execution, similar to Bash's `set -x`, and the NCBImeta[6] project, a command-line application for retrieving and organizing metadata from the NCBI. For instance, in the PySnooper project [44], users attempting to install the package with `pip install pysnooper` encounter a `FileNotFoundError`. The error occurs because the `setup.py` script tries to read a *requirements.txt* file that does not exist, halting the installation process with the output in Figure 4.2.1.

Another example of a missing requirements.txt file is the NCBImeta project [38]. When users run `pip install NCBImeta` to install the package, the installation fails due to the missing *requirements.txt* file, producing error in Figure 4.2.2.

These cases illustrate how the absence of a *requirements.txt* file disrupts the

[5] `https://pypi.org/project/PySnooper/`
[6] `https://pypi.org/project/NCBImeta/`

standard installation workflow, preventing users from setting up the project without additional intervention. The setup scripts in both projects expect the file to specify dependencies, and its absence triggers immediate failures, underscoring the file's critical role in Python package management.

Beyond these installation issues, the lack of a *requirements.txt* file introduces further complications. Without it, developers must manually examine the project's source code to identify all imported external packages—a task that becomes increasingly difficult in large or complex projects with numerous files and dependencies. Even after identifying the packages, determining the correct versions remains a significant hurdle. Packages evolve over time, with newer versions potentially introducing breaking changes or removing features that the project relies upon. Without documentation of the project's intended dependency versions—typically provided by a *requirements.txt* file—developers must resort to trial and error to find compatible versions, which can result in runtime errors or unexpected behavior. Additionally, packages often have interdependencies, requiring careful resolution of version conflicts. These challenges emphasize the need for an automated tool capable of inferring both the required packages and their compatible versions based on the project's context.

4.3 Proposed Solution and Methodology

The absence of a *requirements.txt* file, as discussed in the previous section, leads to significant challenges such as installation failures and manual dependency identification. To address these issues, we introduce a standalone generator tool that automates the creation of a *requirements.txt* file by parsing project code and querying historical version data from PyPI, guided by the project's release and last update dates. This tool operates independently of SMTpip (Chapter 3), which focuses on resolving dependency conflicts for an existing *requirements.txt* file. The generator produces a *requirements.txt* file that serves as an input to SMTpip, forming a complementary pipeline where the generator creates the dependency specification and SMTpip ensures a conflict-free environment.

The methodology consists of the following steps:

1. **Parsing Import Statements**: The generator scans all Python files in the project directory to extract import statements using static code analysis. It identifies external packages (e.g., `numpy`, `requests`) while excluding standard library modules. The tool supports various import patterns, such as `import pandas` or `from sklearn import metrics`, ensuring a comprehensive list of dependencies.

2. **Collecting Project Metadata**: The generator requires the project's release date and last update date, provided by the user. These dates define the temporal range for selecting library versions. For example, a project released on March 1, 2021, and last updated on July 10, 2023, will have its version selection of packages set to those available between these dates.

3. **Querying PyPI's Historical Data**: The generator queries PyPI's release history to retrieve version information for each library. It assigns a version range where the lower bound is the latest version available before the release date, and the upper bound is the latest version before the last update date. For instance, if `requests` had version 2.25.0 available in February 2021 and 2.28.0 in June 2023, the generator specifies `requests >= 2.25.0, <= 2.28.0` in the *requirements.txt* file.

4. **Generating the Requirements File**: The generator compiles the identified packages and their version ranges into a *requirements.txt* file, formatted for compatibility with `pip`. An example entry might be:

```
numpy >= 1.19.0, <= 1.23.0
```

The generator's role concludes with the creation of the *requirements.txt* file. It does not perform dependency conflict resolution or ensure compatibility with the target Python version. Instead, the generated file serves as input to SMTpip, which applies

dependency constraint solving to produce a conflict-free environment. The generator's workflow, illustrated in Figure 4.3.1, relies solely on code parsing and PyPI's historical data, minimizing manual effort and errors. This pipeline—where the generator creates the *requirements.txt* file and SMTpip refines it—ensures reliable project deployment across systems.

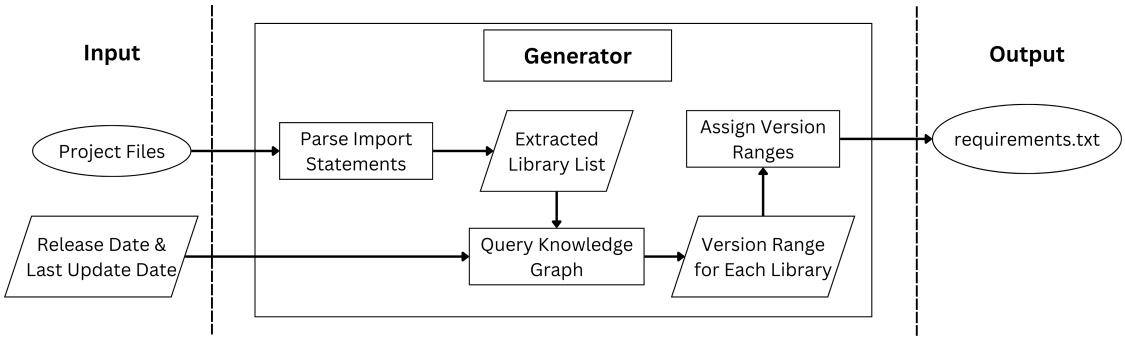


Fig. 4.3.1: Requirements.txt generator workflow.

When combined with SMTpip's conflict resolution, it achieves the ultimate goal of enabling reproducible, conflict-free environments for Python projects lacking dependency specifications.

4.4 Evaluation and Results

Having outlined the methodology for generating *requirements.txt* files in the previous section, we now evaluate the effectiveness of our standalone generator tool for Jupyter notebook projects. This section addresses the following research question (RQ):

RQ: Can our generator tool accurately generate missing *requirements.txt* files?

4.4.1 Evaluation Procedure

To assess the generator tool's ability to infer dependencies and reconstruct missing *requirements.txt* files, we applied it to 1,359 Jupyter Notebook projects (comprising 3,081 *.ipynb* files), as described in Section 3.4.1. For each project, we removed the existing *requirements.txt* file to recreate the missing requirements.txt scenario and assigned the generator to reconstruct dependencies by analyzing import statements

and determining version ranges based on the project's release and last update dates, as detailed in Section 4.3. The generator produces a *requirements.txt* file, which is then used by SMTpip (Chapter 3) to resolve dependency conflicts and create a conflict-free environment.

A baseline comparison was conducted with `pipreqs`,[7] a widely used tool for generating *requirements.txt* files. We selected `pipreqs` because it is a standard, popular choice for automatically scanning project files to identify imported packages and generate a *requirements.txt* file, typically using the latest available package versions. This makes it a suitable benchmark for evaluating our generator's performance. According to its PyPI description, `pipreqs` does not consider temporal constraints or project metadata, which can lead to incompatible dependencies.

To validate accuracy, we created isolated virtual environments for each project using SMTpip with the generated *requirements.txt* files and executed all notebooks. A *successful execution* was defined as completing all notebook cells without dependency-related errors.

4.4.2 Results

Results, presented in Table 4.4.1, show that the pipeline achieved a 39.92% success rate, with 1,230 of 3,081 notebooks executing fully. In contrast, `pipreqs` achieved a 20.84% success rate (642 notebooks), significantly lower than our pipeline's performance. The improved success rate of our approach stems from the generator's use of temporal constraints (i.e., selecting package versions available during the project's active development period) and SMTpip's conflict resolution, which ensures compatibility with the target Python version.

These results demonstrate that our generator, combined with SMTpip, effectively synthesizes *requirements.txt* files for projects lacking dependency specifications, outperforming `pipreqs` by nearly 2×. The pipeline—where the generator creates the *requirements.txt* file and SMTpip resolves conflicts—addresses the challenges of missing dependency specifications.

[7] `https://pypi.org/project/pipreqs/`

Table 4.4.1: Success Rate Comparison: SMTpip vs. pipreqs

Tool	Successful Notebooks	Success Rate
SMTpip	1,230	39.92%
pipreqs [42]	642	20.84%

Error Type	Frequency
ModuleNotFoundError	592
FileNotFoundError	518
TypeError/SyntaxError	461
Environment-Specific	280

4.4.3 Failure Analysis

Failures were primarily due to non-dependency-related issues, detailed below with examples.

ModuleNotFoundError (32%) occurs when a module is imported but not installed, often because static analysis cannot detect dynamic imports. For instance, a notebook using `importlib.import_module('seaborn')` failed with *ModuleNotFoundError: No module named 'seaborn'*. Similarly, another notebook imported `sklearn.metrics`, but the generated *requirements.txt* included only `sklearn` without specifying a version that included `metrics`, resulting in *ModuleNotFoundError: No module named 'sklearn.metrics'*.

FileNotFoundError (28%) is triggered when referenced files are missing. For example, a notebook attempting to load a dataset with `pd.read_csv('data.csv')` failed when `data.csv` was absent, producing *FileNotFoundError: [Errno 2] No such file or directory: 'data.csv'*. Another notebook using `plt.imread('image.png')` failed due to a missing image file, resulting in *FileNotFoundError: [Errno 2] No such file or directory: 'image.png'*.

TypeError/SyntaxError (25%) arises from incompatible API usage or syntax issues. For instance, a notebook using an outdated `numpy` API, such as `numpy.random.RandomState(0)`, with a newer version where the API changed, failed with *TypeError: RandomState() got an unexpected keyword argument 'dtype'*. Another notebook with Python 2 syntax in a Python 3 environment, like `print 'Hello'`, caused *SyntaxError: Missing parentheses in call to 'print'. Did you mean print('Hello')?*.

Environment-Specific Issues (15%) involve problems related to hardware or OS dependencies. For example, a notebook requiring `pycuda` failed on a system without a compatible GPU, producing *RuntimeError: No CUDA GPUs are available*. Another notebook using a Windows-specific library on a Linux system failed with *ImportError: DLL load failed: The specified module could not be found*.

RQ: Can Generator Tool Generate Missing Requirements.txt Files?

In particular:

- The generator tool successfully generated *requirements.txt* files and successfully executed 1,230 out of 3,081 notebooks, achieving a **39.92% success rate** in recreating executable environments.
- Compared to `pipreqs`, which achieved a **20.84% success rate** (642 notebooks), SMTpip demonstrated nearly 2× better performance due to its contextual version resolution and temporal constraint handling.
- Failures were primarily due to non-dependency issues, such as **ModuleNotFoundError** (32%), **FileNotFoundError** (28%), **TypeError/SyntaxError** (25%), and **Environment-Specific Issues** (15%), highlighting challenges beyond dependency resolution.

4.5 Related Work

The evaluation in the previous section demonstrated the effectiveness of our generator tool and its pipeline with SMTpip in reconstructing missing *requirements.txt* files for Jupyter notebook projects. To contextualize our contribution, this section reviews existing tools and research related to dependency management and automated configuration synthesis in Python projects.

Several tools address the generation of *requirements.txt* files by analyzing Python code. `pipreqs`, as discussed in Section 4.4, scans project files to identify imported packages and generates a *requirements.txt* file using the latest available package versions.

While effective for simple projects, `pipreqs` does not consider the project's development timeline, often leading to incompatible versions, as evidenced by its 20.84% success rate in our evaluation. Similarly, `pipdeptree`[8] visualizes dependency trees and can export them as *requirements.txt* files, but it requires an existing environment and does not infer versions for projects lacking dependency specifications. `poetry`[9] and `conda`[10] offer advanced dependency management but rely on pre-existing configuration files (e.g., `pyproject.toml` or `environment.yml`), making them unsuitable for legacy projects without such files.

Research on automated dependency inference has explored static and dynamic analysis techniques. For instance, Cheng et al. [10] proposed a method to infer dependencies by analyzing import statements and runtime behavior, but it requires executing the code, which is impractical for notebooks with missing data files or environment-specific dependencies (e.g., `pycuda`. Other studies, such as smartPip [47], focus on version constraint solving but assume an existing *requirements.txt* file, aligning more closely with SMTpip's role than our generator's. Unlike these approaches, our generator leverages PyPI's historical data and temporal constraints to reconstruct *requirements.txt* files without requiring code execution, making it suitable for legacy and poorly documented projects like those in the PySnooper and NCBImeta cases [44, 38].

Dependency conflict resolution, addressed by SMTpip, has parallels in tools like `pip`'s internal resolver and `poetry`'s dependency solver. However, these tools operate on existing dependency specifications, whereas SMTpip integrates with our generator's output, forming a pipeline that handles both missing configurations and conflict resolution. This combination distinguishes our work, as it addresses the full spectrum of challenges—from generating missing *requirements.txt* files in the format shown in Figure 4.1.1 to ensuring a conflict-free environment.

[8] `https://pypi.org/project/pipdeptree/`
[9] `https://python-poetry.org/`
[10] `https://docs.conda.io/`

4.6 Threats to Validity

The related work section positioned our generator and SMTpip within the landscape of dependency management tools and research. Here, we discuss potential threats to the validity of our evaluation (Section 4.4) to provide a balanced perspective on the results and limitations of our approach.

Internal Validity: The accuracy of the generated *requirements.txt* files depends on the completeness of static code analysis. 32% of failures were due to `ModuleNotFoundError` from undetected dynamic imports (e.g., `importlib.import_module('seaborn')`). Static analysis cannot capture runtime imports, which may lead to incomplete dependency lists. Additionally, the reliance on user-provided release and last update dates introduces potential errors if these dates are inaccurate, as they directly influence version range selection. For example, an incorrect release date might result in a *requirements.txt* file with incompatible versions, such as those seen in the NCBImeta installation failure [38].

External Validity: Our evaluation focused on 1,359 Jupyter Notebook projects, which may not represent all Python project types (e.g., web applications, CLI tools). Jupyter notebooks often rely on data files and environment-specific dependencies (e.g., `pycuda`), contributing to 28% `FileNotFoundError` and 15% environment-specific failures. Generalizing our findings to non-notebook projects could be limited, as these may have different import patterns or dependency structures. However, the generator's methodology (Figure 4.3.1) is agnostic to project type, suggesting potential applicability beyond notebooks.

4.7 Conclusion and Future Work

The previous section highlighted threats to validity, underscoring the challenges of evaluating dependency generation for Python projects. This chapter introduced a standalone generator tool to address the second research problem: generating missing *requirements.txt* files for Python projects, particularly legacy or poorly documented

ones like PySnooper and NCBImeta [44, 38]. The generator automates dependency inference by parsing import statements and querying PyPI's historical data, guided by project release and last update dates, as outlined in Section 4.3. It produces a *requirements.txt* file, formatted as shown in Fig. 4.1.1, which serves as input to SMTpip (Chapter 3) for dependency conflict resolution.

Our evaluation on 1,359 Jupyter Notebook projects (Section 4.4) demonstrated a 39.92% success rate, with 1,230 of 3,081 notebooks executing successfully, nearly doubling the performance of `pipreqs` (20.84%). Failures due to `ModuleNotFoundError`, `FileNotFoundError`, `TypeError/SyntaxError`, and environment-specific issues highlight limitations beyond dependency management, such as dynamic imports and missing data files.

Future work could enhance the generator by incorporating dynamic analysis to detect runtime imports, reducing `ModuleNotFoundError` occurrences. Integrating version inference with code execution traces could improve version accuracy, addressing `TypeError/SyntaxError` issues from API changes. Expanding the knowledge graph to include metadata from GitHub repositories or PyPI download statistics could refine temporal constraints. Additionally, validating the generated *requirements.txt* files in virtual environments before SMTpip's resolution could catch errors early. Combining these improvements with SMTpip's conflict resolution would further advance the pipeline's ability to support diverse Python projects, ensuring seamless deployment and maintenance.

This work represents a significant step toward automating dependency management for Python projects lacking *requirements.txt* files. By providing a standalone generator and a complementary conflict resolver, we address critical challenges in software reproducibility, paving the way for more reliable and maintainable Python ecosystems.

CHAPTER 5

Conclusion

This chapter concludes the thesis by summarizing the key findings and contributions. Section 5.1 provides a summary of the research, while Section 5.2 discusses potential future research directions in the field.

5.1 Summary of Research Findings and Contributions

This thesis addresses the pervasive challenges of dependency conflicts and version incompatibilities in modern Python development, which hinder reproducibility, disrupt workflows, and compromise system stability. We introduced SMTpip, a novel dependency resolution framework that leverages Satisfiability Modulo Theories (SMT) to generate conflict-free installation plans. By integrating a dynamically constructed dependency knowledge graph with constraint-driven optimization, SMTpip translates dependency rules, Python version constraints, and user requirements into SMT expressions. This approach supports both configuration-aware resolution for projects with dependency files and code-driven inference for those without, ensuring broad applicability across diverse Python projects.

Our comprehensive evaluation spanned four datasets: three benchmark datasets (Watchman, HG2.9K, SD) and a newly curated collection of 1,359 real-world Jupyter Notebook projects. The results demonstrate SMTpip's superior performance, as it successfully resolved all dependency conflicts and version incompatibilities in consistent cases while accurately identifying inconsistent cases. Compared to state-of-the-art

tools like pip, Conda, smartPip, and PyEGo, SMTpip achieved remarkable efficiency, resolving dependencies up to 39× faster than pip. These outcomes highlight the power of SMT solvers in navigating Python's complex dependency ecosystem.

The key contributions of this work are threefold:

1. **SMT-driven Dependency Resolution Framework**: SMTpip provides a robust and efficient solution for dependency management, ensuring compatibility during installation.

2. **Public Dataset of Real-world Dependency Conflicts**: The curated dataset of 1,359 Jupyter Notebook projects serves as a valuable resource for advancing research on reproducibility and dependency management.

3. **Empirical Validation of SMT in Dependency Management**: Through rigorous evaluation, we demonstrated the viability of formal methods in addressing real-world software engineering challenges, bridging the gap between installation-time conflict resolution and post-hoc environment repair.

5.2 Future Research Directions

While SMTpip represents a significant advancement in Python dependency management, several opportunities exist to extend and refine its capabilities. One promising direction is to adapt the SMTpip framework to other programming ecosystems, such as JavaScript (npm), Java (Maven), or R (CRAN). These ecosystems face similar dependency challenges, and applying SMT-based resolution could yield comparable benefits. However, this would require constructing ecosystem-specific dependency knowledge graphs and adapting SMT expressions to account for unique versioning conventions and dependency structures.

Another area for exploration is the integration of incremental resolution strategies to handle evolving dependencies. As projects receive updates, their dependency requirements may change, necessitating re-resolution of installation plans. Developing mechanisms to incrementally update the dependency knowledge graph and SMT

constraints, rather than recomputing them from scratch, could significantly reduce resolution times and improve scalability for large, dynamic projects.

Finally, exploring the application of machine learning to predict dependency conflicts or recommend optimal version combinations could complement SMTpip's formal methods. By training models on historical dependency data, such as the public dataset introduced in this work, future iterations of SMTpip could proactively identify potential issues before resolution begins, further streamlining the development process.

The success of SMTpip underscores the transformative potential of formal methods in software engineering. By addressing the limitations of existing tools and laying the groundwork for future innovations, this work paves the way for more robust, efficient, and scalable dependency management across diverse software ecosystems.

REFERENCES

[1] Abate, P., Cosmo, R. D., Gousios, G., and Zacchiroli, S. (2020). Dependency solving is still hard, but we are getting better at it. In *Proceedings of the 27th International Conference on Software Analysis, Evolution and Reengineering*, pages 547–551.

[2] Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., and Zohar, Y. (2022). cvc5: A versatile and industrial-strength SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442.

[3] Barrett, C., Sebastiani, R., Seshia, S., and Tinelli, C. (2009). Satisfiability modulo theories. In *Handbook of Satisfiability*, volume 185, pages 825–885.

[4] Barrett, C., Stump, A., and Tinelli, C. (2010). The SMT-LIB standard: Version 2.0. In *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories*.

[5] Biere, A., Faller, T., Fazekas, K., Fleury, M., Froleyks, N., and Pollitt, F. (2024). CaDiCaL 2.0. In *Proceedings of the 36th International Conference on Computer Aided Verification*, page 133–152.

[6] Biere, A., Heule, M., van Maaren, H., and Walsh, T. (2009). Handbook of satisfiability. In *Handbook of Satisfiability*, volume 185.

[7] Bjørner, N. S. and Phan, A.-D. (2014). νZ - maximal satisfaction with Z3. In *International Symposium on Symbolic Computation in Software Science*, pages 1–9.

[8] Bright, C., Gerhard, J., Kotsireas, I., and Ganesh, V. (2020). Effective problem solving using SAT solvers. In *Maple in Mathematics Education and Research*, page 205–219.

[9] Cao, Y., Chen, Z., Zhang, X., Li, Y., Chen, L., and Wang, L. (2024). Diagnosis of package installation incompatibility via knowledge base. In *Science of Computer Programming*, volume 235, page 103098.

[10] Cheng, W., Hu, W., and Ma, X. (2024). Revisiting knowledge-based inference of Python runtime environments: A realistic and adaptive approach. In *IEEE Transactions on Software Engineering*, pages 258–279.

[11] Cheng, W., Zhu, X., and Hu, W. (2022). Conflict-aware inference of Python compatible runtime environments with domain knowledge graph. In *Proceedings of the 44th International Conference on Software Engineering*, pages 451–461.

[12] Claes, M., Mens, T., Cosmo, R. D., and Vouillon, J. (2015). A historical analysis of Debian package incompatibilities. In *Proceedings of the International Conference on Mining Software Repositories*, pages 212–223.

[13] Conda (2024). Conda documentation. `https://docs.conda.io/en/latest/`. Last Accessed: 2024-09-18.

[14] Conda Community (2020). Understanding and improving Conda's performance. `https://www.anaconda.com/blog/understanding-and-improving-condas-performance`. Last Accessed: 2024-10-12.

[15] Cox, R. (2016). Version SAT. `https://research.swtch.com/version-sat`. Last Accessed: 2024-10-12.

[16] Davis, M., Logemann, G., and Loveland, D. (1962). A machine program for theorem-proving. In *Communications of the ACM*, volume 5, pages 394–397.

[17] de Moura, L. and Bjørner, N. (2008). Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340.

[18] Decan, A., Mens, T., and Claes, M. (2017). An empirical comparison of dependency issues in OSS packaging ecosystems. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, pages 2–12.

[19] Decan, A., Mens, T., Claes, M., and Grosjean, P. (2016). When GitHub meets CRAN: An analysis of inter-repository package dependency problems. In *Proceedings of the International Conference on Software Analysis, Evolution, and Reengineering*, volume 1, pages 493–504.

[20] Dilhara, M., Ketkar, A., and Dig, D. (2021). Understanding software-2.0: A study of machine learning library usage and evolution. In *ACM Transactions on Software Engineering and Methodology*, volume 30, pages 1–42.

[21] Fan, G., Wang, C., Wu, R., Xiao, X., Shi, Q., and Zhang, C. (2020). Escaping dependency hell: finding build dependency errors with the unified dependency graph. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 463–474.

[22] Fflpy (2024). issue #1: Dependency conflict in fflpy repository. `https://github.com/smkell/fflpy/issues/1`. Last Accessed: 2024-10-07.

[23] Gario, M. and Micheli, A. (2015). pySMT: A library for SMT formulae manipulation and solving. In *Proceedings of the 4th International Symposium on International Conference on Computer Aided Verification*, pages 360–368.

[24] Gousios, G. (2013). The GHTorent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 233–236.

[25] Horton, E. and Parnin, C. (2018). Gistable: Evaluating the executability of Python code snippets on GitHub. In *Proceedings of the International Conference on Software Maintenance and Evolution*, pages 217–227.

[26] Horton, E. and Parnin, C. (2019a). DockerizeMe: Automatic inference of environment dependencies for Python code snippets. In *Proceedings of the 41st International Conference on Software Engineering*, pages 328–338.

[27] Horton, E. and Parnin, C. (2019b). V2: Fast detection of configuration drift in Python. In *Proceedings of the 34th International Conference on Automated Software Engineering*, pages 477–488.

[28] Huang, K., Chen, B., Shi, B., Wang, Y., jian Xu, C., and Peng, X. (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*.

[29] Islam, M. J., Nguyen, G., Pan, R., and Rajan, H. (2019). A comprehensive study on deep learning bug characteristics. In *Proceedings of the International Conference on Software Engineering*, pages 510–520.

[30] Li, Z., Wang, Y., Lin, Z., Cheung, S.-C., and Lou, J.-G. (2022). NUFIX: escape from NuGet dependency maze. In *Proceedings of the International Conference on Software Engineering*, pages 1545–1557.

[31] Liang, J. H., Ganesh, V., Poupart, P., and Czarnecki, K. (2016). Learning rate based branching heuristic for SAT solvers. In *Proceedings of the 19th International Conference for Theory and Applications of Satisfiability Testing*, pages 123–140.

[32] Liu, C., Chen, S., Fan, L., Chen, B., Liu, Y., and Peng, X. (2022). Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem. In *Proceedings of the International Conference on Software Engineering*, pages 672–684.

[33] Lopez, J., Kapfhammer, G. M., and McMinn, P. (2012). On the application of SAT solvers to test suite minimization. In *Proceedings of the 4th International Symposium on Search Based Software Engineering*, pages 246–251.

[34] Mamba (2020). Making Conda fast again. `https://wolfv.medium.com/making-conda-fast-again-4da4debfb3b7`. Last Accessed: 2024-10-12.

[35] Masina, G., Spallitta, G., and Sebastiani, R. (2023). On CNF conversion for disjoint SAT enumeration. In *Proceedings of the 26th International Conference on Theory and Applications of Satisfiability Testing*.

[36] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. (2001). Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535.

[37] Mukherjee, S., Almanza, A., and Rubio-González, C. (2021). Fixing dependency errors for Python build reproducibility. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 439–451.

[38] NCBImeta (2018). Ncbimeta issue #10: Missing "requirements.txt" in PyPI package. `https://github.com/ktmeaton/NCBImeta/issues/10`. Last Accessed: 2024-10-13.

[39] Nimble (2021). Modern techniques for dependency resolution. `https://nimblepkg.github.io/docs/dependency-resolution`. Last Accessed: 2024-10-12.

[40] PEP440 (2013). Version identification and dependency specification. `https://peps.python.org/pep-0440/#version-specifiers`. Last Accessed: 2024-10-12.

[41] pip (2020). pip documentation v24.2. `https://pip.pypa.io/en/stable/topics/dependency-resolution/`. Last Accessed: 2024-10-10.

[42] pipreqs (2024). Generate requirements.txt file for any project based on imports. `https://github.com/bndr/pipreqs`. Last Accessed: 2024-10-13.

[43] PyPI (2024). Python package index. `https://PyPI.org/`. Last Accessed: 2024-09-18.

[44] PySnooper (2018). Pysnooper issue #26: pip install fails for missing requirements.txt. `https://github.com/cool-RR/PySnooper/issues/26`. Last Accessed: 2024-10-13.

[45] Soto-Valero, C., Harrand, N., Monperrus, M., and Baudry, B. (2021). A comprehensive study of bloated dependencies in the Maven ecosystem. In *Proceedings of Empirical Software Engineering*, pages 123–135.

[46] Vouillon, J. and Cosmo, R. D. (2013). On software component co-installability. In *ACM Transactions on Software Engineering and Methodology*, volume 22, pages 34:1–34:35.

[47] Wang, C., Wu, R., Song, H., Shu, J., and Li, G. (2022). smartPip: A smart approach to resolving Python dependency conflict issues. In *Proceedings of the 37th International Conference on Automated Software Engineering*.

[48] Wang, H., Liu, S., Zhang, L., and Xu, C. (2023). Automatically resolving dependency-conflict building failures via behavior-consistent loosening of library version constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

[49] Wang, J., Li, L., and Zeller, A. (2021). Restoring execution environments of Jupyter notebooks. In *Proceedings of the 43rd International Conference on Software Engineering*, pages 1622–1633.

[50] Wang, Y., Wen, M., Liu, Y., Wang, Y., Li, Z., Wang, C., Yu, H., Cheung, S.-C., Xu, C., and Zhu, Z. (2020). Watchman: Monitoring dependency conflicts for Python library ecosystem. In *Proceedings of the 42nd International Conference on Software Engineering*, pages 125–135.

[51] Wang, Y., Wen, M., Liu, Z., Wu, R., Wang, R., Yang, B., Yu, H., Zhu, Z., and Cheung, S.-C. (2018). Do the dependency conflicts in my project matter? In *Proceedings of the 26th ACM Joint Meeting on European Software Engineering*

Conference and Symposium on the Foundations of Software Engineering, pages 319–330.

[52] Ye, H., Chen, W., Dou, W., Wu, G., and Wei, J. (2022). Knowledge-based environment dependency inference for Python programs. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1245–1256.

[53] Zhu, C., Saha, R. K., Prasad, M., and Khurshid, S. (2021). Restoring the executability of Jupyter notebooks by automatic upgrade of deprecated APIs. In *Proceedings of the 36th International Conference on Automated Software Engineering*, pages 240–252.

[54] Zhu, R., Wang, X., Liu, C., Xu, Z., Shen, W., Chang, R., and Liu, Y. (2024). ModuleGuard: understanding and detecting module conflicts in Python ecosystem. In *Proceedings of the 46th International Conference on Software Engineering*, pages 1–12.

APPENDIX A

Sample SMT Instances

This appendix presents sample SMT instances generated by SMTpip, smartPip, and PyEGo for dependency resolution, focusing on the packages `python-dateutil` and `six`. Each tool encodes package version constraints differently, as shown in the simplified snippets below. Full expressions are omitted for brevity. The snippets illustrate variable declarations, constraints, and optimization objectives, highlighting the structural differences in their approaches.

A.1 SMTpip: Boolean Variables

SMTpip declares a Boolean variable for each package version and the `none` case, ensuring mutual exclusivity with `(_ at-most 1)` constraints. Soft assertions with weights optimize for newer versions or non-installation.

```
; Variable Declarations (for python-dateutil)
(declare-fun python-dateutil==none () Bool)
(declare-fun python-dateutil==2.7.3 () Bool)
(declare-fun python-dateutil==2.8.2 () Bool)
; ... (other versions)

; Constraint: At most one version
(assert ((_ at-most 1)
  python-dateutil==none
  python-dateutil==2.7.3
  python-dateutil==2.8.2
```

```
  ; ... (other versions)))

; Exclusion Constraints
(assert (=> python-dateutil==none (not python-dateutil==2.7.3)))
(assert (=> python-dateutil==none (not python-dateutil==2.8.2)))

; Optimization (soft assertions)
(assert-soft python-dateutil==none :weight 1)
(assert-soft python-dateutil==2.8.2 :weight 0.833)
```

A.2 smartPip: Integer Variables with Nested Logical Constraints

smartPip assigns integer variables to packages, mapping versions to numerical values. Constraints enforce valid version ranges, and multiple maximization objectives aim to favor newer versions, though Z3 prioritizes only the first objective.

```
; Variable Declaration
(declare-fun python-dateutil () Int)

; Constraints: Version Range and Specific Values
(assert (< python-dateutil 2726797)) ; Upper bound
(assert (>= python-dateutil 2726794)) ; Lower bound
(assert (or (= python-dateutil 2726794) ; 2.7.3
            (= python-dateutil 2726796))) ; 2.8.2

; Dependency Constraint (e.g., with six)
(assert (or (and (= python-dateutil 2726796) (< six 9232) (>= six 9215))
            ; ... (other conditions)))

; Optimization
```

```
(maximize python-dateutil)
; ... (other maximize directives)
```

smartPip uses multiple maximization functions (e.g., *(maximize python-dateutil)*, *(maximize six)*) to prefer newer versions. However, Z3 only allows one effective optimization objective, executing the first one fully and disregarding subsequent ones.

A.3 PyEGo: Nested Logical Constraints

PyEGo uses integer variables and complex logical expressions (`Or`, `And`) to model dependencies, ensuring compatibility across version combinations. It issues multiple `(check-sat)` calls for different constraint types.

```
(set-logic QF_LIA)

; Dependency Constraint (for six and python)
(assert (Or (And (= six 8)
                 (Or (= python 0) (= python 1) (= python 2)))
            (And (= six 1)
                 (Or (= python 0) (= python 1) (= python 2)))
            ; ... (other combinations)))

; Integer Constraint
(assert (And (six >= 0) (six <= 11)))

; Version Constraint
(assert (Or (six == 0) (six == 1) (six == 8) ; ... (valid versions)))

(check-sat)
```

VITA AUCTORIS

NAME:	Sadman Jashim Sakib
PLACE OF BIRTH:	Dhaka, Bangladesh
YEAR OF BIRTH:	1998
EDUCATION:	BRAC University, B.Sc in Computer Science & Engineering, Dhaka, Bangladesh, 2022
	University of Windsor, M.Sc in Computer Science, Windsor, Ontario, 2025