SMTpip: Interpreter-Aware SMT-Based Dependency Conflict Resolution for Restoring Python Source-Code Executability

Sadman Jashim Sakib
University of Windsor
Windsor, Canada
sakib51@uwindsor.ca

Muhammad Asaduzzaman
University of Windsor
Windsor, Canada
Muhammad.Asaduzzaman@uwindsor.ca

Curtis Bright
University of Waterloo
Waterloo, Canada
cbright@uwaterloo.ca

***Abstract*—Software developers rely on packages to reuse existing functionality instead of implementing everything from scratch. Python developers commonly provide package and interpreter dependencies using configuration files, such as *requirements.txt* or *setup.py*. Package managers in Python, such as pip, can install packages according to dependency and interpreter version constraints specified in configuration files. However, Python dependency resolution remains challenging: (1) different packages may require incompatible versions of the same dependency; (2) dependencies may require a Python interpreter version that is incompatible with the interpreter used for the project, making a valid environment impossible; and (3) pip, the most popular Python package manager, resolves conflicts via backtracking, repeatedly trying candidate versions without knowing whether a valid execution environment exists or not. To address these challenges, we present SMTpip, an interpreter-aware environment inference technique for improving the executability of Python source-code artifacts. SMTpip constructs a dependency knowledge graph using metadata stored in the Python Package Index (PyPI) that hosts millions of package releases, encodes both package version constraints and interpreter compatibility constraints specified in configuration files into Satisfiability Modulo Theories (SMT) formulas. Solving these formulas identifies a set of package versions and an interpreter version that jointly satisfy all declared constraints. Empirical evaluation on multiple datasets from open-source Python projects shows that SMTpip achieves substantial speedups—6.9× over pip, 9.6× over Conda, 3.2× over smartPip, and 4× over PyEGo—while consistently producing constraint-consistent environments. Our execution-based evaluation shows that SMTpip enables more successful executions and substantially fewer interpreter-related failures than baseline tools. The tool and datasets are publicly available in a replication package.**

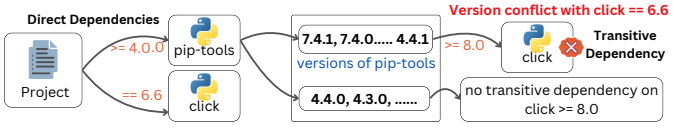
***Index Terms*—Python, SMT Solver, Dependency Management, Dependency Conflict, Conflict Resolution, Software Management.**

I. Introduction

Software developers rely on third-party packages to reuse functionality and accelerate development. In Python projects, dependency requirements are recorded in configuration files such as *requirements.txt* and *setup.py* (Figure 3). These files serve as a blueprint for recreating the software environment by specifying required packages and version constraints. Python packages are distributed through the Python Package Index (PyPI) [1], a central repository that hosts third-party packages. Package managers (such as *pip* [2] and *Conda* [3]) use these configuration files to download and install specified packages and any other packages that are dependent on them maintaining their version constraints. However, the installation of packages may be affected by dependency conflict issues when multiple version constraints are specified for the same package, potentially leading to build failures.

For example, installing the Python project *fflpy* can lead to a dependency conflict (issue #1 [4]). During installation of dependent packages exactly one version per required package must be selected such that all direct and transitive version constraints, as well as Python interpreter compatibility constraints, are satisfied. Figure 1 illustrates a dependency conflict in this setting: the project requires *click* == 6.6 and also requires *pip-tools* ≥ 4.0.0, but newer *pip-tools* releases (e.g., 7.4.1) introduce a transitive constraint *click* ≥ 8.0, making those candidate versions of *pip-tools* incompatible with the project's pinned *click* requirement. In this scenario, pip first installs *click* version 6.6 as specified in the configuration file. Next, pip attempts to install the latest available *pip-tools* version (7.4.1) as it meets the version constraint (*pip-tools* ≥ 4.0.0). However, *pip-tools* version 7.4.1 has a dependency on *click* ≥ 8.0. Since the project explicitly requires *click* == 6.6, pip detects a dependency conflict and backtracks to the previous candidate version (7.4.0). Pip continues this trial-and-error process over other candidate versions until it reaches *pip-tools* == 4.4.0, which does not impose a conflicting constraint on *click*, thereby resolving the dependency conflict. However, pip does not know in advance how many candidate versions it needs to try, how much computation is required, and whether a valid execution environment exists.

Existing package managers can struggle with handling dependency conflicts. For example, pip can take a significantly long time to resolve a dependency conflict as shown in the previous example. Conda is another widely used package manager (popular in scientific computing) that also performs dependency resolution. However, during our evaluation we found that Conda's classic solver configuration incurs substantial overhead in resolving dependency conflicts. We use the classic solver because it employs SAT-based dependency




```
Collecting click==6.6
Collecting pip-tools>=4.0.0
INFO: pip is looking at multiple versions...
Downloading pip-tools-7.4.1-py3-none-any.whl (0.24 MB)
pip-tools 7.4.1 requires click>=8.0
ERROR: Cannot install pip-tools==7.4.1 and click==6.6
Downloading pip-tools-7.4.0-py3-none-any.whl (0.23 MB)
pip-tools 7.4.0 requires click>=8.0
ERROR: Cannot install pip-tools==7.4.0 and click==6.6
Downloading pip-tools-7.3.0-py3-none-any.whl (0.22 MB)
pip-tools 7.3.0 requires click>=8.0
ERROR: Cannot install pip-tools==7.3.0 and click==6.6
... (multiple repeated attempts) ...
Downloading pip-tools-4.4.0-py3-none-any.whl (0.15 MB)
Successfully installed pip-tools-4.4.0 click-6.6
```

Fig. 1: An example of a dependency conflict and how pip resolves it using backtracking.

resolution, making it a directly relevant baseline for SMTpip's CNF-based formulation. Conda also provides other solver implementations, such as libmamba, but evaluating alternative solvers is outside the scope of this study. Conda's developers have also acknowledged that, in practice, Conda can be slower than pip for dependency resolution [5]–[8]. Several prior studies also address dependency conflict issues. For example, PyEGo [9] focuses on executability by inferring environments for Python scripts, but it does not comprehensively resolve transitive dependency conflicts at install time (and may still rely on pip's backtracking during installation). SmartPip [10] uses a pre-built knowledge base and *satisfiability modulo theories* (SMT) solving (i.e., constraint solving over logical formulas with background theories), but its constraints are not expressed in conjunctive normal form (CNF). Because modern SMT solvers internally require CNF encodings, non-CNF encodings require a translation step that introduces overhead. SmartPip also generally installs extra packages that are not needed.

To address these challenges and to support the executability of Python source code, we propose **SMTpip**, an SMT-based dependency conflict resolution technique that (i) builds a dependency knowledge graph from PyPI metadata, (ii) encodes dependency and interpreter constraints directly in a compact CNF form, and (iii) uses a weighted Max-SMT objective to select preferred solutions (e.g., favoring newer compatible versions and avoiding unnecessary installations). Direct CNF generation avoids relying on internal CNF transformations that introduce auxiliary variables; additional variables require the solver to perform additional propagation, the cost of which dominates SAT/SMT solving time [11], [12]. SMTpip avoids the repeated trial-and-error process inherent in pip's backtracking-based dependency conflict resolution. Our evaluation spans four different datasets: three of those datasets are collected from previous studies (WatchMan [13], HG2.9K [14], and SD [9]) and a new dataset of 1,359 real-world Jupyter Notebook projects curated from GitHub. Across these datasets, SMTpip achieves substantial speedups over existing tools (6.9× over pip, 9.6× over Conda, 3.2× over smartPip, and 4× over PyEGo) in dependency resolution time (RQ2), while maintaining strong resolution coverage (RQ1). In addition, we validate whether the environments generated by SMTPip improve the executability of source code (RQ3). Results from the study show that environments produced by SMTpip enable more successful program executions and substantially fewer interpreter-related failures than baseline tools.

This paper makes the following contributions:

- **SMTpip.** A Python dependency resolver that computes an *executable environment* by jointly selecting compatible package versions and a Python *interpreter version* that satisfy all declared constraints, or reports when no such environment exists due to conflicting constraints. It uses a direct CNF encoding with a weighted Max-SMT formulation to efficiently optimize version selection and improve executability.
- **Datasets.** A new benchmark of 1,359 GitHub Jupyter Notebook projects, alongside three established datasets (WatchMan, HG2.9K, and SD).
- **Evaluation.** A comparative evaluation against pip, Conda's classic solver, smartPip, and PyEGo, including execution-based validation of resolved environments.
- **Tool and artifacts.** We publicly release **SMTpip**, along with all datasets and evaluation artifacts required to reproduce our results. The tool and replication package are available on Zenodo [15], [16].

The remainder of this paper is organized as follows. Section II introduces background and formal definitions. Section III positions SMTpip relative to prior work. Section IV presents our approach and encoding. Section V reports experimental results for RQ1–RQ3. Section VI discusses questions related to our study and Section VII discusses threats to validity. Finally, Section VIII concludes the paper.

II. Preliminaries

This section provides necessary background related to our study.

A. Packages, Versions, and Constraints

For each package p, let V_p denote the finite set of available version numbers of package p.

a) Version constraints: Say a package has a dependency on package q. A *version constraint* C_q on q defines a subset $S(C_q) \subseteq V_q$ specifying the versions of q meeting the dependency. In practice, constraints are derived from PEP 440 version specifiers such as inclusive bounds ($\leq$, $\geq$), exclusive bounds ($<$, $>$), and the compatible release operator $\sim=$.

b) Third-party package dependency constraints (direct and transitive): For a package version p_v ($v \in V_p$), let Q be the set of packages p_v depends on, and C_q ($q \in Q$) the version constraint that p_v has on package q. Then $\{ S(C_q) : q \in Q \}$ denotes its dependency specification, where each subset $S(C_q)$ denotes that selecting p_v requires a package $q_{v'}$ to be selected where $v' \in S(C_q)$ for each $q \in Q$. Dependencies are *transitive* when they are induced indirectly via other dependencies rather than listed directly by the client project.

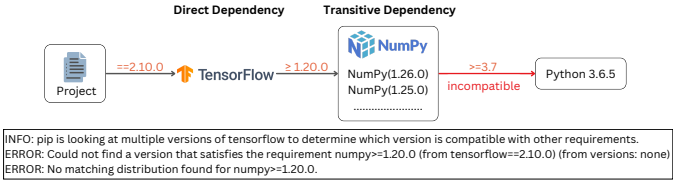


Fig. 2: An example of installation failure due to Python version incompatibilities.

setup.py

```
import setuptools
setuptools.setup(
    name='fflpy'
    version='0.0.1',
    description='fflpy Project',
    packages=['fflpy'],
    python_requires='>=3.6',
    install_requires=['click==6.6',
                      'pip-tools>=4.0.0'])
```

requirements.txt

```
click==6.6
pip-tools>=4.0.0
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
```

Fig. 3: Examples of configuration files.

c) Interpreter constraints: Let $\mathcal{I}$ denote the set of available Python interpreter versions. Each package version p_v may additionally define an interpreter compatibility constraint specifying the subset of Python interpreter versions $\psi(p_v) \subseteq \mathcal{I}$ that can be used with package p_v.

B. Dependency Conflict

a) Third-party package conflict: A third-party package conflict may occur in the process of selecting a set of package versions meeting all dependency requirements simultaneously. A *dependency conflict* happens when a set of selected package versions requires a package q to be installed, but there is no consistent way of selecting a package version of q meeting all dependency constraints simultaneously.

For example, in Figure 1, the project has constraints *click* == 6.6 and *pip-tools* ≥ 4.0.0. Candidate versions of *pip-tools* such as 7.4.1 impose a transitive constraint *click* ≥ 8.0, making those candidates conflicting with *click* == 6.6. However, older candidates (e.g., *pip-tools* == 4.4.0) remain compatible, so the overall instance is satisfiable even though several candidate versions of pip-tools will result in a dependency conflict if they are installed.

b) Interpreter compatibility conflict: An *interpreter compatibility conflict* occurs when an interpreter constraint causes there to be no consistent way of installing packages meeting the constraint. For example, in Figure 2 a project requires *TensorFlow* == 2.10.0, which depends on *numpy* ≥ 1.20.0. However, all versions of *numpy* ≥ 1.20.0 require *Python* ≥ 3.7.0. Therefore, if the environment uses Python 3.6.5, no version of NumPy can satisfy both the dependency and interpreter constraints, and the installation fails.

C. Dependency Resolution and Environment Setup

Dependency resolution selects exactly one version for each required package, and one Python interpreter version I, so that: (i) all direct constraints are satisfied, (ii) all transitive dependency constraints induced by selected versions are satisfied, and (iii) all interpreter constraints hold, i.e., $I \in \psi(p_v)$ for all selected package versions p_v. When flexible version specifiers yield many package version candidates, dependency resolution solvers must identify and eliminate conflicting package versions until they find a constraint-consistent environment (or determine that no such environment exists).

D. Satisfiability Solving

Satisfiability solving asks a simple question: *can we assign true/false values to variables so that a logical formula becomes true?* A standard normal form for logical formulae is called **Conjunctive Normal Form (CNF)**. A CNF formula is a conjunction (AND, $\wedge$) of clauses, where each clause is a disjunction (OR, $\vee$) of literals, and a literal is either a variable or its negation. For example, $(x \vee \neg y) \wedge (\neg x \vee z)$ is in CNF. CNF is the standard input format for modern SAT solvers, as modern SAT solving algorithms operate directly on clauses. Non-CNF encodings can be converted into CNF by introducing new variables [17], but this increases the formula size, and in general smaller CNF encodings with fewer variables tend to give better solver performance [12]. The notation $x \rightarrow y$ is shorthand for $\neg x \vee y$.

III. Related Work

Prior work on Python environment setup and dependency management spans several research directions. We group the related works into three different categories: Dependency Resolving, Build-failure Repair, and SAT/SMT in Software Engineering.

Dependency Resolving. A prominent line of work builds a dependency knowledge base and applies reasoning to select compatible package versions. **DockerizeMe** [18] combines static analysis of code snippets with PyPI metadata and heuristics to assemble runnable environments, but its inferred dependency graph can be incomplete. **SmartPip** [10] strengthens this direction by constructing a PyPI-based knowledge graph and using an SMT solver to search for compatible versions from configuration files; however, smartPip uses a non-CNF encoding and does not explicitly compute a globally compatible Python interpreter version, which can contribute to interpreter-related failures. **PyEGo** [9] uses an SMT solver together with code-based inference to recover environments for single-file scripts, but it is not designed for multi-file projects or configuration-driven dependency resolution; furthermore, it may leave transitive dependency conflict resolution to pip during installation. Other techniques such as **PyCRE** [19] and **PCREQ** [20] focus on extracting or reconstructing requirement sets, but they do not directly target efficient conflict resolution under Python's single-version semantics. More broadly, Pinckney et al. formulate dependency management using SMT/Max-SMT and graph-based reasoning in **PACSOLVE** [21]; however, this work primarily targets ecosystems such as NPM where multiple versions of a package can be co-installed. This avoids many single-version conflicts, but introduces a different dependency-graph selection problem. Among these approaches, **smartPip**

and **PyEGo** are the most relevant to our work because they also use solver-based reasoning to select dependency versions.

Build-failure Repair. Another set of approaches attempts installation and uses build/installer feedback to propose fixes. **ReadPyE** [22] extends PyEGo with log-driven validation to iteratively repair failing setups. **PyDFix** [23] and pip-error mining approaches (e.g., Cao et al. [24]) parse installer outputs to identify missing modules and propose dependency patches, while **V2** [25] combines a failure database with guided search over candidate versions. These methods can be effective in practice, but they often incur substantial overhead due to repeated failed installs and retries, and may struggle with nested transitive conflicts or ambiguous error messages. Complementary work addresses failures that occur after installation, such as API deprecations or missing runtime resources. **SnifferDog** [26] targets notebooks by inferring dependencies from cells and traces, **RELANCER** [27] focuses on notebook API deprecations, and **LooCo** [28] explores constraint relaxation guided by code evidence. Outside Python, systems such as **Decca** [29] and **LibHarmo** [30] harmonize versions using API-compatibility signals and cross-project usage, and SAT-based harmonization techniques have been explored (e.g., [31]). Recent work also explores large language models: **PLLM** [32] combines retrieval-augmented generation with an error-driven loop to suggest fixes, but such approaches are not fully deterministic and may hallucinate or suggest incorrect updates without structured validation. Since build failures can be caused by dependency conflicts, techniques that focus on build-failure repair can benefit from SMTpip by using its constraint-consistent dependency resolution to avoid or reduce trial-and-error installation attempts.

SAT/SMT in Software Engineering. SAT/SMT solvers have also been widely applied beyond dependency management, including program analysis and verification, test generation, fault localization, and security analysis (e.g., [33]–[36]). However, the objectives of these techniques are different from ours.

IV. SMTpip: SMT-driven approach

Figure 4 shows the overview of our proposed approach, SMTpip. The approach consists of two parts: knowledge graph construction and dependency resolution. 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 an SMT encoding.

A. 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. 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.

a) Data collection: Figure 4 illustrates the data collection procedure. PyPI, 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 retrieved 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.

b) 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.

c) Knowledge graph representation of package dependencies: The dependency constraints were modeled into a knowledge graph representing relationships between packages, their versions, and respective dependencies. The graph comprises two types of nodes: **package nodes**, representing individual packages, and **version nodes**, representing specific versions of a package. Each version node includes two properties—*python_version*, denoting the required Python version, and *release_date*, indicating the version's publication date on PyPI. The graph also includes two types of edges: **versioning edges**, which link package nodes to their version nodes, and **dependency edges**, which connect version nodes of dependent packages to capture inter-version dependencies. Collectively, these components form a comprehensive structure modelling the hierarchical relationships and dependencies within the Python package ecosystem. Stored in JSON format (122 MB), the knowledge graph supports efficient analysis and resolution of third-party package dependency conflicts and Python version incompatibilities.

d) Knowledge graph update: Downloading the entire PyPI repository from scratch for every update is neither practical nor scalable due to the continuous growth and frequent changes in PyPI. To support efficient maintenance, SMTpip implements an incremental update mechanism that leverages the `last_serial` value provided by PyPI [37]. Instead of re-fetching metadata for all packages, the updater identifies packages whose metadata has changed since the last update and retrieves only the modified records. Using this approach, a complete update scan over the entire PyPI ecosystem requires approximately 3.2 hours. Furthermore, SMTpip provides a project-level refresh mechanism through the command `python SMTpip.py -d project_directory --refresh`, which checks PyPI only for the packages used within the target project and updates their local metadata before dependency resolution. This operation typically completes in less than one second and propagates any retrieved updates to the shared knowledge graph. For reproducibility, the generated graph is published as a JSON artifact in a GitHub repository; however, production deployments can utilize more scalable storage backends such as indexed databases or object stores. The knowledge graph could also be hosted in a server, instead of across users, avoiding the need for each developer to maintain

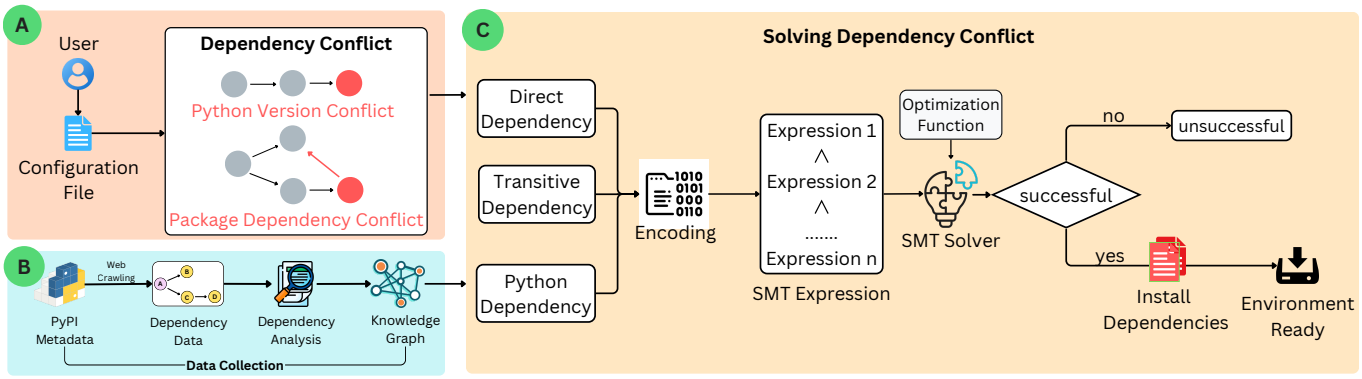


Fig. 4: SMTpip architecture

a separate 122 MB copy.

B. 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 I. In this section, we formalize the process of encoding the dependency resolution problem as a satisfiability problem.

Given a set of dependency constraints (as shown in Figure 3), 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, we define the literal p_v representing that version v of package p is to be installed. Say $V(p)$ denotes the set of all version numbers of 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 [38] supports the parameterized theory function "`(_ at-most 1)`" as part of an extension of the SMT-LIB [39].

Secondly, we need to express the requirements found in the configuration file (e.g., *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 $p \geq 2$ would only be satisfied by 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 an acceptable version of p will be installed with 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 the starting set may 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 encoded similarly.

Finally, we would like the solver to prioritize the most recent versions of packages (when possible) and to not install any unnecessary packages. In order to do this, we define an objective function that the solver maximizes. This function assigns higher values to solutions selecting more recent versions and encourages not installing packages at all. 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 $p_{\text{none}} = 1$ gives the highest possible value of 1 in the parenthesized expression in the objective function. 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.

C. Example of SMT Encoding of Dependency Resolution

Consider the open-source project fflpy [4], with the dependencies *click* == 6.6 and *pip-tools* ≥ 4.0.0. We encode these dependencies along with all transitive dependencies and Python version requirements using Boolean variables and clauses. Let c_v be a Boolean variable for version v of *click* (with c_{none} denoting no version of *click* will be installed), and p_v be a Boolean variable for version v of *pip-tools* (with v_{none} denoting no version of *pip-tools* will be installed). Let A be the set of all *click* versions (and the symbol none), B be the set of all *pip-tools* versions (and the symbol none), and $T \subseteq B$ be the versions of *pip-tools* ≥ 4.0.0, e.g., $\{7.4.1, 7.4.0, \ldots, 4.0.0\}$.

a) Package Requirement Constraints: *click* == 6.6 is encoded as $c_{6.6}$, and *pip-tools* ≥ 4.0.0 is encoded as $\bigvee_{v \in T} p_v$ (e.g., $p_{7.4.1} \vee p_{7.4.0} \vee \cdots \vee p_{4.0.0}$).

b) Consistency Constraints: The constraint at most one version per package is selected is encoded as $\sum_{v \in A} c_v \leq 1 \wedge \sum_{v \in B} p_v \leq 1$. 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 $v \in B \setminus \{\text{none}\}$.

c) Transitive Dependency Constraints:

- *pip-tools* == 7.4.1 requires *click* ≥ 8.0 encoded as $p_{7.4.1} \rightarrow (c_{8.0} \vee c_{8.1})$
- *pip-tools* == 4.4.1 requires *click* ≥ 7.0 encoded as $p_{4.4.1} \rightarrow (c_{8.0} \vee c_{8.1} \vee c_{7.0} \vee c_{7.1})$

d) Python Version Constraints:

- *click* == 6.6 requires *python* ≥ 2.7 encoded as $c_{6.6} \rightarrow (python_{2.7} \vee python_{2.8} \vee \cdots)$
- *pip-tools* == 7.4.1 requires *python* ≥ 3.9 encoded as $p_{7.4.1} \rightarrow (python_{3.9} \vee python_{3.10} \vee \cdots)$

e) Optimization: To encode the objective function in SMT, we formulate a MaxSMT problem using the following weighted soft constraints. A constraint is soft if it is not strictly required, but the solver tries to satisfy as many soft constraints as possible.

- 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. Moreover, the dependency constraints are expressed directly in CNF. The consistency constraints are expressed using cardinality constraints, but SMT solvers are designed to automatically convert cardinality constraints into CNF efficiently.

f) Solving and Validation: Once all dependency constraints are encoded into SMT expressions, we utilize the Z3 SMT solver [38] 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.

After obtaining the solution (satisfying assignment), a validation step is performed using the knowledge graph. In this step, all dependencies of the selected versions of the packages in the solution set are cross-verified to ensure the packages selected to install do indeed satisfy all requirements and do not introduce any conflicts when installed together. This validation check is not strictly required, but acts as a sanity check that the correct SMT encoding was provided to the SMT solver. In cases where no satisfying assignment exists, the solver produces a proof of the inconsistency of the instance.

V. Evaluation

This section discusses the evaluation procedure and results of our study.

A. Datasets

For the purpose of this study, we consider four different datasets (see Table I). Three of those datasets (Watchman [13], HG2.9K [14], and SD [9]) were collected from prior studies [9], [10], [18], [22]. We created a fourth dataset, consisting of 1,359 real-world Jupyter Notebook projects collected from GitHub, specifically curated to capture a diverse range of dependency structures found in real-world scenarios.

The Notebook dataset was derived from the GHTorent archive [40] and initially contained 247,356 Python-based Jupyter Notebook projects. We queried the GitHub API to identify projects containing *requirements.txt* files, a standard indicator of dependency specifications. This filtering step reduced the dataset to 22,340 projects with dependency metadata. We then used pip to install dependencies for each project and recorded the installation logs. We analyzed the logs to detect instances in which pip invoked backtracking during dependency resolution, which we used as an indicator of dependency conflicts. The resulting Notebook Dataset comprises 1,359 real-world Jupyter Notebook projects with dependency conflicts. In total, the dataset contains 3,081 *.ipynb* files.

We also measured the size of the dependency graph for each case across all datasets. The size of a dependency

TABLE I: Datasets used for evaluation and information about the instances in each dataset.

Dataset	Total Cases	Dependency Graph Size (Min/Max/Median/Average)
WatchMan	159	8 / 85 / 30 / 13
HG2.9K	2891	1 / 99 / 17 / 6
SD	100	1 / 85 / 11 / 4
Notebook	1359	2 / 161 / 10 / 20

graph is defined as the total number of unique packages that the project depends on, directly or indirectly. For example, consider a project that depends on *dagit* version 1.1.5 which in turn depends on *dagster* version 1.1.5, and *dagster* depends on *universal-pathlib* (any version). If there are no other dependencies, the dependency graph size will be 3. The total number of unique packages in this graph represents its size. The maximum dependency graph sizes observed in our datasets are 85, 99, 73, and 161 for the Watchman, HG2.9K, SD, and Notebook datasets, respectively.

B. RQ1: How effective is SMTpip in resolving third-party package dependency conflicts and Python version incompatibilities?

This section evaluates the performance of **SMTpip** in resolving third-party package dependency conflicts and Python version incompatibilities during installation, compared to four existing tools: pip, Conda, smartPip, and PyEGo. Results are presented in two parts: the first part assesses the tools' ability to resolve dependency conflicts using the *latest dependency knowledge graph*, as shown in Table II to compare with pip, Conda, and PyEGo. The replication package of smartPip relies on a dependency knowledge base that was last updated in 2022. Since dependency resolution may be affected by the availability of newer dependency metadata, using more recent information could introduce bias in the comparison. To ensure a fair comparison with smartPip, we restrict the dependency information used in our experiments to versions available up to 2022, aligning with the temporal scope of SMTpip's knowledge base. Thus, the second part discusses the evaluation conducted using a *downgraded knowledge graph* to compare with smartPip, as shown in Table III. In the tables, the column *Total Time (sec)* represents the time required by each tool to perform dependency resolution for all projects in each benchmark. This measurement excludes time spent downloading packages or performing installations, as those operations are not considered part of the resolution process itself.

SMTpip demonstrates superior performance in resolving dependency conflicts, as evidenced in Table II and Table III. Using the latest knowledge graph, SMTpip achieves resolution rates of **72%, 58%, 76%, and 66%** across WatchMan, HG2.9K, SD, and Notebook datasets, respectively. For example, in the HG2.9K dataset, SMTpip resolves 1,668 cases in **433.68 seconds**. In contrast, pip, while achieving 58% resolution for HG2.9K (1,668/2,891), requires **3,185.20 seconds** due to exhaustive backtracking and repeated metadata downloads. Conda resolves only 37% of HG2.9K cases (1,062/2,891) in **7,516.80 seconds**, as multiple SAT solver invocations for optimization increase runtime. PyEGo, limited to HG2.9K and SD due to its focus on single-file scripts, resolves 49% of HG2.9K cases (1,435/2,891) in **1,817.40 seconds**. Its heuristic pre-selection of popular packages can prematurely eliminate valid dependency combinations, limiting applicability to multi-file projects and configuration files.

In the comparison with smartPip using a downgraded knowledge graph, SMTpip resolves more cases and does so more efficiently (Table III). For example, on HG2.9K, SMTpip resolves 1,656/2,891 cases (58%) in **263.36 seconds**, whereas smartPip resolves 1,640/2,891 cases (56%) in **1,482.06 seconds**. A reason for smartPip's lower success rate is that it intentionally narrows the set of candidate versions by internally rewriting some constraints: in particular, it replaces $\geq$ and $>$ constraints with the compatible-release operator $\sim=$, which permits only minor-version upgrades. For instance, suppose a project requires a dependency version constraint $\geq 1.4.5$, but smartPip internally treats it as $\sim= 1.4.5$. Under PEP 440 [41], $\sim= 1.4.5$ restricts the selected version to the range $\geq 1.4.5$ and $< 1.5.0$, thereby excluding versions that still satisfy the original $\geq 1.4.5$ constraint. The original constraint does not impose this upper bound; thus, smartPip rejects valid versions 1.5.0 and higher.

To better understand the unresolved cases, we manually analyzed SMTpip failures and categorized them into four groups. *UNSAT Constraints* correspond to projects whose dependency and Python-version requirements are inherently contradictory, meaning that no valid environment exists. Resolving such cases would require modifying the project's requirements (e.g., upgrading or downgrading dependency constraints), which is beyond the scope of SMTpip and the compared techniques. SMTpip can report unsatisfiable dependency configurations to developers to help identify conflicting requirements; validating these conflicts directly with package developers is left for future work. *Missing Metadata* occurs when the dependency knowledge graph lacks required dependency information for a package version due to incomplete metadata available from PyPI. *Unsupported pip Features* account for approximately 5% of unresolved cases and arise when projects rely on pip-specific functionality that is not yet supported by SMTpip, such as platform-specific dependency markers, direct URL dependencies, or local path dependencies. Finally, *Solver Timeout* occurs when the SMT solver cannot complete within the imposed 500-second timeout because of extremely large dependency graphs and search spaces.

We also observed that SMTpip successfully resolves all cases that can be resolved by pip, Conda, smartPip, or PyEGo. Across all evaluated datasets, we did not encounter any instance where SMTpip failed while another tool succeeded. Even in benchmarks where SMTpip and pip resolve a similar number of projects, SMTpip demonstrates substantially better robustness on difficult dependency-resolution tasks. Pip's backtracking-based search strategy may struggle or time out when exploring very large dependency search spaces. For example, pip timed out on two projects in the WatchMan dataset and four projects in the Notebook dataset. One representative example is a

project depending on `tableschema-py`, where SMTpip successfully resolved the dependency conflict and produced a valid environment in 22.41 seconds, whereas pip exceeded the 500-second timeout limit without finding a solution. These results demonstrate that SMTpip can efficiently handle challenging real-world dependency conflicts that are difficult for trial-and-error based dependency resolution approaches.

In addition to finding a compatible assignment when one exists, SMTpip explicitly reports when the constraint set is *inconsistent* (i.e., no version selection can satisfy all declared dependency and interpreter constraints). This diagnostic is useful because it indicates that the installation failure is caused by mutually contradictory constraints that must be fixed upstream (e.g., by package maintainers or by correcting project requirements), rather than by trying additional candidate versions. In contrast, tools such as pip may expend substantial effort exploring candidates before failing, without clearly distinguishing an inherent inconsistency from search exhaustion; SMTpip returns an explicit UNSAT result for inconsistent instances.

TABLE II: Results of SMTpip, pip, Conda, and PyEGo in resolving dependency conflicts using latest dependency knowledge graphs.

Dataset	Tool	Resolved Dependencies	
		Num	Total Time (sec)
WatchMan	pip	112/159 (70%)	946.40 s
	Conda	31/159 (19%)	352.22 s
	SMTpip	**114/159 (72%)**	**226.86 s**
HG2.9K	pip	1668/2891 (58%)	3185.20 s
	Conda	1062/2891 (37%)	7516.80 s
	PyEGo	1435/2891 (49%)	1817.40 s
	SMTpip	**1668/2891 (58%)**	**433.68 s**
SD	pip	76/100 (76%)	253.50 s
	Conda	24/100 (24%)	482.8 s
	PyEGo	62/100 (62%)	84.52 s
	SMTpip	**76/100 (76%)**	**36.48 s**
Notebook	pip	887/1359 (65%)	5289.50 s
	Conda	784/1359 (58%)	2414.90 s
	SMTpip	**891/1359 (66%)**	**400.95 s**

TABLE III: Results of SMTpip and smartPip in resolving dependency conflicts using downgraded dependency knowledge graphs.

Dataset	Tool	Resolved Dependencies	
		Num	Total Time (sec)
WatchMan	smartPip	104/159 (65%)	533.50 s
	SMTpip	**110/159 (69%)**	**156.22 s**
HG2.9K	smartPip	1640/2891 (56%)	1482.06 s
	SMTpip	**1656/2891 (58%)**	**263.36 s**
SD	smartPip	62/100 (62%)	39.20 s
	SMTpip	**68/100 (68%)**	**24.42 s**
Notebook	smartPip	689/1359 (50%)	393.03 s
	SMTpip	**711/1359 (53%)**	**305.73 s**

C. RQ2: Is the Technique Efficient Enough for Practical Use?

The efficiency of SMTpip in resolving package dependency conflicts is critical to its suitability for practical use in real-world software development. Thus, we analyze the speedup achieved by SMTpip compared to four established tools—pip, Conda, smartPip, and PyEGo—across four datasets: WatchMan, HG2.9K, SD, and Notebook. Table IV presents a comprehensive time comparison for projects where both SMTpip and the compared tool successfully resolved dependency conflicts. Here, Success Count (SC) represents the number of projects where the dependency conflict was resolved by both tools, Time Cost (TC) denotes the total dependency-resolution time, and Speedup is computed as the ratio of the compared tool's time cost to SMTpip's.

Table IV shows that SMTpip consistently achieves substantial speedups across all datasets. Overall, SMTpip is 6.9× faster than pip, 9.6× faster than Conda, 3.2× faster than smartPip, and 4× faster than PyEGo on the combined datasets. For example, on HG2.9K, SMTpip resolves 1,668 projects in 433.68 seconds, compared to pip's 2,165.20 seconds (4.9× speedup). Pip's longer times are due to iterative backtracking, which may explore many candidate versions before converging. Conda exhibits significant delays as well: on HG2.9K, SMTpip is 16.5× faster (4,566.23 seconds vs. 276.12 seconds for the 1,062 projects where both succeed), reflecting the additional overhead of Conda's optimization criteria and repeated solver invocations.

In comparison with smartPip, SMTpip achieves a 5.6× speedup on HG2.9K (1,476 seconds vs. 262.4 seconds for the 1,640 projects where both succeed). SmartPip's higher cost is due to the non-CNF encoding that increases solver overhead. PyEGo, evaluated only on HG2.9K and SD due to its focus on single-file Python scripts, shows SMTpip being 4.2× faster on HG2.9K (1,467.4 seconds vs. 346.84 seconds for the 1,334 projects where both succeed). In smaller datasets such as SD, SMTpip's speedup over smartPip is modest (1.1×), reflecting that simpler instances reduce the relative benefit of SMT-based global reasoning.

D. RQ3: Do the environments generated by SMTPip improve the executability of source code?

While RQ1 and RQ2 evaluate resolution success and speed, they do not assess whether the resolved virtual environments are *functionally accurate*—that is, whether the selected dependencies actually allow the project to execute correctly. Therefore, in RQ3 we evaluate the practical executability of the resolved environments. We performed execution evaluation on both the HG2.9K and Notebook datasets because their entry points can be identified reliably. The WatchMan and SD datasets were excluded from execution testing because they consist of multi-file projects, making it difficult to reliably determine the appropriate entry point for automated execution.

For each program gist in the HG2.9K dataset whose environment was successfully resolved by a tool, we created an isolated virtual environment, installed the selected package versions, and executed the program entry point. Execution outcomes were categorized as: (1) successful execution; (2) *Module/API failures*, typically caused by deprecated packages or API-breaking changes; (3) *Interpreter failures*, including installation errors due to incompatible Python versions or `SyntaxError` from unsupported language features; and (4) *Other* failures, including OS-specific dependencies, external

TABLE IV: 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 the 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	112	654.40 s — 226.86 s (**2.8**×)	31	138.22 s — 61.69 s (**2.2**×)	104	520 s — 151.84 s (**3.4**×)	N/A	N/A
HG2.9K	1668	2165.20 s — 433.68 s (**4.9**×)	1062	4566.23 s — 276.12 s (**16.5**×)	1640	1476 s — 262.4 s (**5.6**×)	1435	1467.4 s — 346.84 s (**4.2**×)
SD	76	198.50 s — 36.48 s (**5.4**×)	24	118.20 s — 11.52 s (**10.2**×)	42	25.2 s — 22.9 s (**1.1**×)	62	63.24 s — 29.76 s (**2.12**×)
Notebook	887	4568.50 s — 400.95 s (**11.3**×)	784	1936.50 s — 352.8 s (**5.4**×)	689	378.95 s — 296.27 s (**1.27**×)	N/A	N/A
Sum	**2743**	**7586.6 s — 1097.97 s (6.9×)**	**1901**	**6759.15 s — 702.13 s (9.6×)**	**2475**	**2400.15 s — 733.45 s (3.2×)**	**1396**	**1530.64 s — 376.56 s (4×)**

TABLE V: Execution validation results on HG2.9K and Notebook datasets with unified error categories.

Dataset	Tool	Executed	Success (%)	Module/ API Errors	Interpreter Errors	Other Errors
HG2.9K Dataset						
HG2.9K	pip	1257	75.4%	287	102	22
HG2.9K	smartPip	1235	75.3%	283	98	24
HG2.9K	PyEGo	1331	92.7%	72	26	6
HG2.9K	**SMTpip**	**1453**	**87.1%**	**150**	**37**	**28**
Notebook Dataset						
Notebook	pip	616	20.0%	920	345	1265
Notebook	smartPip	554	18.0%	980	341	1205
Notebook	PyEGo	770	25.0%	810	391	1110
Notebook	**SMTpip**	**1230**	**39.92%**	**592**	**280**	**979**

services, tool-specific integrations, missing packages, or absent datasets and configuration files.

Table V summarizes the results. Although pip and SMTpip resolve the same number of environments (1668), SMTpip yields substantially more successful executions (1453 vs. 1257). This improvement is primarily due to interpreter selection. Because most HG2.9K gists specify dependencies only through *requirements.txt*, Python version requirements are typically absent (Figure 3). As a result, pip resolves dependencies using the currently installed interpreter without searching for alternative Python versions. In contrast, SMTpip incorporates `Requires-Python` metadata and jointly resolves interpreter and package version constraints, enabling the selection of a compatible interpreter. Consequently, SMTpip achieves an execution success rate of 87.1%, compared to 75.4% for pip and 75.3% for smartPip. Although PyEGo attains a higher success rate (92.7%), it resolves fewer environments (1435 vs. 1668), resulting in fewer successful executions overall (1331 vs. 1453).

SMTpip also substantially reduces interpreter-related failures (37 vs. 102 for pip and 98 for smartPip). This reduction is largely due to interpreter handling: when projects are specified only via *requirements.txt*, pip and smartPip resolve dependencies under the already-installed interpreter and do not compute or recommend an alternative Python version; as a result, interpreter incompatibilities may appear as installation failures or `SyntaxError` caused by unsupported language features. Across all tools, most remaining failures stem from module/API mismatches caused by breaking changes in major-version upgrades, while others arise from host-specific libraries, proprietary infrastructure, or missing external artifacts that fall outside the scope of dependency resolution.

We also evaluated SMTpip on the Notebook dataset. Table V summarizes the results. The dataset comprises 1,359 real-world Jupyter Notebook projects with dependency conflicts and contains 3,081 `.ipynb` files, as each project typically includes multiple notebooks. We executed all 3,081 notebooks using environments resolved by all tools. SMTpip achieves the highest execution success rate of 39.92% (1,230/3,081), outperforming pip (20.0%), smartPip (18.0%), and PyEGo (25.0%). The most common failure categories for SMTpip were `Module/API errors` (592 cases), `Interpreter errors` (280 cases), and `Other errors` (979 cases), where the latter includes file-not-found issues, type/syntax errors, and missing external resources such as datasets and configuration files. Similar failure patterns are observed for baseline tools, although SMTpip consistently reduces interpreter-related failures due to improved interpreter selection and constraint handling.

VI. Discussion

This section discusses questions related to our study.

Number of Packages in the Install Scripts: SMTpip applies an optimization function that prunes unnecessary packages from generated install scripts. This pruning step removes packages that are not required to satisfy the project's dependency constraints, producing a more compact install script. The summary statistics of the package counts produced by SMTpip and smartPip in four datasets are reported in Table VI. As shown in Table VI, smartPip consistently produces larger install scripts across all four datasets. Some projects use only Python's standard library (no third-party packages), but we keep them because selecting a compatible Python interpreter can still matter. To illustrate this difference with a concrete example, consider a simple requirements.txt file that specifies only *click* == 6.6 and *pip-tools* ≥ 4.0.0. After dependency resolution, the solver selects *click* == 6.6 and *pip-tools* == 4.2.0, where *pip-tools* == 4.2.0 declares only two runtime dependencies (*click* ≥ 6 and *six*). Consequently, SMTpip produces a minimal install script containing exactly three packages: *click* 6.6, *pip-tools* 4.2.0, and *six* 1.16.0. In contrast, more recent releases of *pip-tools* such as version 7.4.1 declare a substantially larger set of dependencies including *build* ≥ 1.0.0, *click* ≥ 8, *pip* ≥ 22.2, *pyproject-hooks*, *setuptools*, and *wheel*. Although smartPip resolves to the same version *pip-tools* 4.2.0, it expands the install script by including 20 packages, most of which are unnecessary.

Effect of Dependency Graph Size on Performance: We examined whether dependency graph size affects resolution

TABLE VI: Summary statistics of package counts in install scripts produced by SMTpip and smartPip across datasets. "Additional Packages" = smartPip count minus SMTpip count.

Dataset	SMTpip	smartPip	Additional Packages
	Min/Max/Median/Avg	Min/Max/Median/Avg	Min/Max/Median/Avg
Watchman	0/53/3/5.52	0/107/6/11.72	0/54/3/6.20
HG2.9K	0/33/1/2.50	0/76/2/5.66	0/49/1/3.15
Notebook	0/93/1/3.71	0/151/3/8.31	0/74/2/4.61
SD	0/59/6/11.09	0/144/16/24.55	0/90/9/13.46

TABLE VII: Dependency-graph-size groups and timing statistics (seconds).

Quartile	Generating the SMT expression (mean/median/min/max)	Solving the SMT expression (mean/median/min/max)
Q1	0.02 / 0.01 / 0.00 / 0.11	0.00 / 0.00 / 0.00 / 0.02
Q2	0.02 / 0.01 / 0.00 / 0.12	0.01 / 0.01 / 0.00 / 0.03
Q3	0.03 / 0.02 / 0.00 / 0.14	0.17 / 0.07 / 0.03 / 0.57
Q4	0.13 / 0.10 / 0.03 / 0.75	3.50 / 2.82 / 0.45 / 10.01

time using the Notebook dataset, which contains the largest number of projects. The dataset was divided into four quartiles (Q1–Q4) based on dependency graph size, and for each group we measured the time required to generate SMT expressions and to solve them. Table VII summarizes the quartile boundaries and timing statistics.

Generation time increases gradually from smaller graphs (Q1–Q2) to medium graphs (Q3), and then rises sharply for the largest graphs (Q4): the median generation time increases from 0.01 s in Q2 to 0.07 s in Q3 and reaches 2.82 s for Q4 projects. Solving time follows a similar but steeper pattern, with the largest graphs producing the most pronounced increases.

Spearman rank-order correlations confirm these trends. Generation time correlates strongly with dependency graph size, $r_s[1359] = .697$, $p < .001$, while solving time shows an even stronger association, $r_s[1359] = .901$, $p < .001$. These results indicate that larger dependency graphs consistently lead to longer generation and solving times, with solver time exhibiting the dominant growth for larger graphs.

Rationale for Direct CNF Encoding: SMTpip generates SMT constraints directly in conjunctive normal form (CNF), rather than constructing higher-level non-CNF formulas and relying on the solver to perform conversion. This design choice is motivated by both theoretical considerations from SAT/SMT solving and empirical observations. In particular, CNF is expected to perform well because modern SAT/SMT solvers deal with CNF internally. We also evaluated several non-CNF variants and observed consistently worse performance in practice; therefore, we present the CNF encoding used by SMTpip as our primary design. SMTpip uses a compact Boolean encoding with one variable per package–version, enforcing single-version semantics via cardinality constraints and encoding dependencies as implications. Our empirical comparisons against non-CNF variants support this rationale, showing that the CNF encoding used by SMTpip consistently performs best in practice.

VII. Threats to Validity

This section discusses threats to the validity of our research. **External Validity:** Threats to external validity refer to the generalizability of our findings. We evaluate SMTpip using projects written in Python. One can argue that the results may not be generalized to other Python projects. 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. 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.

Internal Validity: Threats to internal validity relate to potential biases or inaccuracies in our methodology and measurements. First, SMTpip relies on the correctness and completeness of package metadata, which may be incomplete or inconsistent in practice. Second, entry-point detection and runtime execution may be affected by environment-specific behavior and nondeterminism (e.g., timeouts or network-dependent code). Third, execution-based evaluation does not guarantee functional correctness, since HG2.9K and Notebook datasets lack ground-truth outputs or test cases, making it impossible to verify program correctness. Finally, results may vary with different solver configurations; we use default Z3 settings for consistency, and all comparisons are performed under identical experimental conditions.

VIII. Conclusion

Dependency conflicts and interpreter incompatibilities in modern Python development hinder reproducibility and slow environment setup. We presented **SMTpip**, an SMT-based framework that models environment setup as a global version-selection problem under both package and interpreter compatibility constraints. SMTpip uses a compact encoding with a weighted Max-SMT formulation to efficiently compute constraint-consistent and executable environments. We evaluated SMTpip on four real-world datasets, including 1,359 GitHub Jupyter Notebook projects and three established benchmarks. SMTpip achieves substantial speedups—6.9× over pip, 9.6× over Conda, 3.2× over smartPip, and 4× over PyEGo—while maintaining strong resolution coverage. Execution-based evaluation further shows that SMTpip produces environments with higher executability and significantly fewer interpreter-related failures compared to baseline tools. Overall, these results demonstrate that interpreter-aware, constraint-based global reasoning can improve both the efficiency and executability of Python dependency resolution in practice. Future work includes extending the framework to ecosystems with different installation semantics and exploring incremental resolution for evolving environments.

References

[1] PyPI, "Python package index," https://PyPI.org/, 2024, Last Accessed: 2024-09-18.

[2] pip, "pip documentation v24.2," https://pip.pypa.io/en/stable/topics/dependency-resolution/, 2020, Last Accessed: 2024-10-10.

[3] Conda, "Conda documentation," https://docs.conda.io/en/latest/, 2024, Last Accessed: 2024-09-18.

[4] Fflpy, "issue #1: Dependency conflict in fflpy repository," https://github.com/smkell/fflpy/issues/1, 2024, Last Accessed: 2024-10-07.

[5] conda1, "Conda issue #7239: Dependency resolution backtracking issues," https://github.com/conda/conda/issues/7239, 2018, Last Accessed: 2024-10-13.

[6] conda2, "Conda issue #8197: Dependency resolution," https://github.com/conda/conda/issues/8197, 2019, Last Accessed: 2024-10-13.

[7] conda3, "Dependency version conflict handling," https://github.com/conda/conda/issues/8810, 2019, Last Accessed: 2024-10-13.

[8] conda4, "Conda issue #9983: Performance improvements for dependency resolution," https://github.com/conda/conda/issues/9983, 2020, Last Accessed: 2024-10-13.

[9] H. Ye, W. Chen, W. Dou, G. Wu, and J. Wei, "Knowledge-based environment dependency inference for Python programs," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 1245–1256. [Online]. Available: https://doi.org/10.1145/3510003.3510127

[10] C. Wang, R. Wu, H. Song, J. Shu, and G. Li, "smartPip: A smart approach to resolving Python dependency conflict issues," in *Proceedings of the 37th International Conference on Automated Software Engineering*, 2022, p. 12. [Online]. Available: https://doi.org/10.1145/3551349.3560437

[11] S. Prestwich, ser. Frontiers in Artificial Intelligence and Applications. IOS Press, 2009, no. 1, pp. 75–97. [Online]. Available: https://doi.org/10.3233/978-1-58603-929-5-75

[12] N. Eén and A. Biere, "Effective preprocessing in SAT through variable and clause elimination," in *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing*, 2005, pp. 61–75. [Online]. Available: https://doi.org/10.1007/11499107_5

[13] Y. Wang, M. Wen, Y. Liu, Y. Wang, Z. Li, C. Wang, H. Yu, S.-C. Cheung, C. Xu, and Z. Zhu, "Watchman: Monitoring dependency conflicts for Python library ecosystem," in *Proceedings of the 42nd International Conference on Software Engineering*, 2020, pp. 125–135. [Online]. Available: https://doi.org/10.1145/3377811.3380426

[14] E. Horton and C. Parnin, "Gistable: Evaluating the executability of Python code snippets on GitHub," in *Proceedings of the International Conference on Software Maintenance and Evolution*, 2018, pp. 217–227. [Online]. Available: https://doi.org/10.1109/ICSME.2018.00031

[15] S. J. Sakib, "SMTpip/SMTpip: Initial release," [tool], Zenodo, v1, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21941892

[16] S. Sakib, "SMTpip/SMTpip_evaluation: Initial release," [evaluation], Zenodo, v1, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21942725

[17] C. Bright, J. Gerhard, I. Kotsireas, and V. Ganesh, "Effective problem solving using SAT solvers," in *Maple in Mathematics Education and Research*, 2020, p. 205–219. [Online]. Available: https://doi.org/10.1007/978-3-030-41258-6_15

[18] E. Horton and C. Parnin, "DockerizeMe: Automatic inference of environment dependencies for Python code snippets," in *Proceedings of the 41st International Conference on Software Engineering*, 2019, pp. 328–338. [Online]. Available: https://doi.org/10.1109/ICSE.2019.00047

[19] W. Cheng, X. Zhu, and W. Hu, "Conflict-aware inference of Python compatible runtime environments with domain knowledge graph," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 451–461. [Online]. Available: https://doi.org/10.1145/3510003.3510078

[20] H. Lei, G. Xiao, Y. Liu, and Z. Zheng, "PCREQ: Automated inference of compatible requirements for Python third-party library upgrades," *ACM Transactions on Software Engineering and Methodology*, 2026. [Online]. Available: https://doi.org/10.1145/3806394

[21] D. Pinckney, F. Cassano, A. Guha, J. Bell, M. Culpo, and T. Gamblin, "Flexible and optimal dependency management via max-SMT," in *Proceedings of the 45th International Conference on Software Engineering*, 2023, pp. 1418–1429. [Online]. Available: https://doi.org/10.1109/ICSE48619.2023.00124

[22] W. Cheng, W. Hu, and X. Ma, "Revisiting knowledge-based inference of Python runtime environments: A realistic and adaptive approach," *IEEE Transactions on Software Engineering*, pp. 258–279, 2024. [Online]. Available: https://doi.org/10.1109/TSE.2023.3346474

[23] S. Mukherjee, A. Almanza, and C. Rubio-González, "Fixing dependency errors for Python build reproducibility," in *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021, pp. 439–451. [Online]. Available: https://doi.org/10.1145/3460319.3464797

[24] Y. Cao, Z. Chen, X. Zhang, Y. Li, L. Chen, and L. Wang, "Diagnosis of package installation incompatibility via knowledge base," *Science of Computer Programming*, vol. 235, 2024. [Online]. Available: https://doi.org/10.1016/j.scico.2024.103098

[25] E. Horton and C. Parnin, "V2: Fast detection of configuration drift in Python," in *Proceedings of the 34th International Conference on Automated Software Engineering*, 2019, pp. 477–488. [Online]. Available: https://doi.org/10.1109/ASE.2019.00052

[26] J. Wang, L. Li, and A. Zeller, "Restoring execution environments of Jupyter notebooks," in *Proceedings of the 43rd International Conference on Software Engineering*, 2021, pp. 1622–1633. [Online]. Available: https://doi.org/10.1109/ICSE43902.2021.00144

[27] C. Zhu, R. K. Saha, M. Prasad, and S. Khurshid, "Restoring the executability of Jupyter notebooks by automatic upgrade of deprecated APIs," in *Proceedings of the 36th International Conference on Automated Software Engineering*, 2021, pp. 240–252. [Online]. Available: https://doi.org/10.1109/ASE51524.2021.9678889

[28] H. Wang, S. Liu, L. Zhang, and C. Xu, "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*, 2023, pp. 198–210. [Online]. Available: https://doi.org/10.1145/3611643.3616264

[29] Y. Wang, M. Wen, Z. Liu, R. Wu, R. Wang, B. Yang, H. Yu, Z. Zhu, and S.-C. Cheung, "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*, 2018, pp. 319–330. [Online]. Available: https://doi.org/10.1145/3236024.3236056

[30] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "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*, 2020, pp. 518–529. [Online]. Available: https://doi.org/10.1145/3368089.3409689

[31] P. Abate, R. D. Cosmo, G. Gousios, and S. Zacchiroli, "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*, 2020, pp. 547–551. [Online]. Available: https://doi.org/10.1109/SANER48275.2020.9054837

[32] A. Bartlett, C. Liem, and A. Panichella, "The last dependency crusade: Solving python dependency conflicts with llms," in *Proceedings of the 40th IEEE/ACM International Conference on Automated Software Engineering Workshops*, 2025, pp. 66–73. [Online]. Available: https://doi.org/10.1109/ASEW67777.2025.00022

[33] Y. Xie and A. Aiken, "Scalable error detection using boolean satisfiability," in *Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, 2005, pp. 351–363. [Online]. Available: https://dl.acm.org/doi/10.1145/1040305.1040334

[34] N. Bjørner, "Engineering theories with Z3," in *Proceedings of the First International Conference on Certified Programs and Proofs*, 2011, pp. 4–16. [Online]. Available: https://doi.org/10.1007/978-3-642-25379-9_1

[35] X. Si, X. Zhang, R. Grigore, and M. Naik, "Maximum satisfiability in software analysis: Applications and techniques," in *Proceedings of the International Conference on Computer Aided Verification*, 2017, pp. 68–94. [Online]. Available: https://doi.org/10.1007/978-3-319-63387-9_4

[36] J. Vanegue, S. Heelan, and R. Rolles, "SMT solvers for software security," in *Proceedings of the 6th USENIX Conference on Offensive Technologies*, 2012, pp. 9–22. [Online]. Available: https://dl.acm.org/doi/10.5555/2372399.2372412

[37] S. J. Sakib, "SMTpip/knowledgegraphupdater: Initial release," [tool], Zenodo, v1, 2026. [Online]. Available: https://doi.org/10.5281/zenodo.21944395

[38] L. de Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, 2008, pp. 337–340. [Online]. Available: https://doi.org/10.1007/978-3-540-78800-3_24

[39] SMT-LIB, "SMT-LIB: The satisfiability modulo theories library," https://smt-lib.org/, 2026, Last Accessed: 2026-08-15.
[40] G. Gousios, "The GHTorent dataset and tool suite," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, 2013, pp. 233–236. [Online]. Available: https://doi.org/10.1109/MSR.2013.6624034
[41] PEP440, "Version identification and dependency specification," https://peps.python.org/pep-0440/#version-specifiers, 2013, Last Accessed: 2024-10-12.