A Probabilistic Approach To Selecting Build Configurations in Package Managers

Daniel Nichols[†], Harshitha Menon[*], Todd Gamblin[§], Abhinav Bhatele[†]

[†]*Department of Computer Science, University of Maryland, College Park, USA*
[*]*Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, USA*
[§]*Livermore Computing, Lawrence Livermore National Laboratory, Livermore, USA*
E-mail: dnicho@umd.edu, {harshitha, tgamblin}@llnl.gov, bhatele@cs.umd.edu

***Abstract*—Modern scientific software in high performance computing is often complex, and many parallel applications and libraries depend on several other software or libraries. Developers and users of such complex software often use package managers for building them. Package managers depend on humans to codify package constraints (for dependency and version selection), and the dependency graph of a software package can often become large (hundreds of vertices). In addition, package constraints often become outdated and inconsistent over time since they are maintained by different people for different packages, which is a laborious task. This can result in package builds to fail for certain package configurations. In this paper, we propose a methodology that uses historical build results to assist a package manager in selecting the best versions of package dependencies with an aim to improve the likelihood of a successful build. We utilize a machine learning (ML) model to predict the probability of build outcomes of different configurations of packages in the Spack package manager. When evaluated on common scientific software stacks, this ML model-based approach is able to achieve a 13% higher success rate in building packages than the default version selection mechanism in Spack.**

***Index Terms*—package managers, build configuration, version selection, machine learning**

I. Introduction

Scientific software has grown in complexity and size over time, and many application codes have tens to hundreds of dependencies on other software and libraries. Keeping track of dependency and version constraints for such complex software and building them manually is highly challenging. As a result, developers and users have turned to package managers to automate the process of managing and installing dependencies. However, package managers depend on humans to codify package constraints (for dependency and version selection). Over time, package constraints can become outdated and inconsistent since maintenance is often laborious and done by different people for different packages.

Package constraints can either be too soft or too strict as developers generally cannot test all of the combinatorial version constraints that arise in large dependency graphs. As package versions and dependency constraints are updated, these invalid constraints can cause packages to fail to build or install. This leads to package managers struggling with building and maintaining packages successfully. Solving these complex package management problems and improving the success rate of package installations is important as the complexity and size of software continues to grow. Thus, ensuring that a package manager can reliably resolve dependency constraints despite the complexity of the package is important. Such an improved package manager could reduce the amount of time developers spend debugging dependency issues and re-building software. This could also reduce the amount of time package maintainers spend writing and maintaining dependency constraints. Finally, this could allow even larger and more complex software to be easily built, maintained, and reliably deployed with a higher success rate.

Ensuring that a package manager builds packages successfully most of the time is difficult as it involves improving the quality of constraints in package definitions (codified by humans). Improving these through brute-force or increased human effort is not feasible due to the large number of packages and dependencies in modern scientific software. Making use of historical build information is a potential solution to this problem, but it is non-trivial to incorporate into modern package managers. This is due to most package managers relying on logic programs and SAT solvers to resolve package configurations, which do not readily support incomplete and/or probabilistic information. Furthermore, trying to adapt these solvers and change their problem encoding is generally a complex task [1]. Minor changes in the problem encoding can lead to significant changes in the solution space that can yield undesirable solutions or be intractable to solve.

In this paper, we improve the success rate of package installations by incorporating historical build information into the version selection process. We first adapt the BuildCheck [2] machine learning model, that predicts the probability of a dependency graph successfully building, and extend it to be able to handle new and unseen versions. We then make use of the predicted probabilities in the flexible and heavily parameterized package manager, Spack [3], to select the versions of a package that maximize the probability of a successful build. We design and compare several strategies for encoding probabilistic information into Spack's internal logic program. We show that our proposed methodology improves the success rate of package installations by up to 13% over the default version selection mechanism in Spack. We further show how this methodology can be used to study the causes of package

SC24, November 17-22, 2024, Atlanta, Georgia, USA
979-8-3503-5291-7/24/$31.00

build failures.

Our paper makes the following important contributions.

- A method for extrapolating build probabilities of package configurations to handle new and unseen versions.
- A novel methodology for selecting package configurations that incorporates probabilistic build information to improve the success rate of package installations.
- A method for exploring the causes of build failures using probabilistic build information.

In addition to these contributions we answer the following research questions in our work:

RQ1 *How can we extrapolate build information to new package versions to mitigate frequent, expensive data collection?* We compare several methods for extrapolating build probabilities on a short time-horizon and show that per-pair averages provide the best results.

RQ2 *How can probabilistic build information be used to select better dependency versions and improve the likelihood of successfully building a package?* We show that incorporating build probabilities into Spack's version selection mechanism with probabilistic Answer Set Programming improves the total number of packages in our data set built successfully by 13%.

RQ3 *Can probabilistic version selection mechanisms be effectively used to determine causality for failed builds?* We demonstrate how to use Plingo, a probabilistic logic programming language, to query build outcome information for to-be installed packages and study causality.

II. Background

In this section, we provide background on the Spack package manager and its use of Answer Set Programming for package configuration selection. We additionally discuss probabilistic variants of Answer Set Programming and machine learning models that yield probabilities of build success for package configurations.

A. Spack and Answer Set Programming

To test our methodologies we use the Spack package manager [3]. It is a flexible and heavily parameterized package manager that allows for the specification of package dependencies, versions, and build flags. It was originally designed for scientific software ecosystems, but has grown to support a wide variety of software. Its parameterization makes it a desirable tool to use for our studies as it allows us to easily modify the package configurations to test different version selections. Spack can also build any of its package configurations which allows us to easily test the build success of different configurations.

Spack packages are defined in a Python based Domain Specific Language (DSL). Despite being in Python, the package DSL is a declarative specification of the package, its build process, dependencies, constraints, and other metadata. Listing 1 shows an example package declaration in Spack.

Spack defines a *concretizer* [4], which is an algorithm that takes an abstract specification of a package and its dependencies and produces a concrete package configuration for building and installing. It is responsible for selecting the versions and build flags for each package in the dependency graph. This concretizer is implemented using formal logic in Answer Set Programming (ASP) [5], [6]. ASP is a declarative programming paradigm that allows for the specification of a problem in terms of rules and constraints. These rules and constraints define *stable models*, which are the solutions to the problem. Internally, Spack uses the Clingo [7] language and solver implementation of ASP.

```
# This is the class name for the package `example`
class Example(Package):
    """Example depends on zlib, mpi,
       and optionally bzip2"""

    version("1.1.0")    # two versions are available
    version("1.0.0")

    variant("bzip", default=True,
            description="enable bzip")

    # Depends on bzip2 or later when bzip is enabled
    depends_on("bzip2@1.0.7:", when="+bzip")
    depends_on("zlib")  # depends on zlib
    # Newer versions require newer versions of zlib
    depends_on("zlib@1.2.8:", when="@1.1.0:")

    # Depends on *some* MPI implementation
    depends_on("mpi")

    # Known failure when building with intel compilers
    conflicts("%intel")
    # Does not support architectures derived from ARM64
    conflicts("target=aarch64:")
```

Listing 1: Constraints in a Spack `package.py` file, expressed in Spack's embedded DSL.

Spack's internal Clingo program consists of two parts: rules that define what a valid package configuration is, and facts that define the package definitions, current build environment, and other metadata. Since there may be many valid package configurations, the concretizer uses *optimization statements* to select the best configuration based on a number of criteria. For instance, the default Spack concretizer prefers the latest version of a package and packages that are already installed.

Listing 2 shows an example of a version conflict specification in the format of Spack's concretizer. This conflict would be generated based on a stated conflict within a package's *package.py* file. If the condition requirements are present in the concretizer's facts for a given package configuration, then an error fact will be included in the model. Spack's concretizer is set to minimize the number of errors in the model, so these conflicts will be avoided if a valid configuration exists. Listing 3 shows the rule that enforces this behavior within Spack's concretizer.

Spack encodes some of its constraints using error facts and aggressively minimizes the number of error facts in the final stable model. This program encoding allows for the propagation of error messages into the final stable model and is essential for providing useful error messages to the user. Line 1 in Listing 3 shows how the *Msg* variable is propagated to the error fact. Rules and constraints in the concretizer that

```
condition(15022,"conflict trigger foo@v1").
condition_requirement(15022,"node","foo").
condition_requirement(15022,"node_version_satisfies",
                      "foo","v1").
condition(15023,"conflict constraint bar@v2").
condition_requirement(15023,"node","bar").
condition_requirement(15023,"node_version_satisfies",
                      "bar","v2").
conflict("foo",15022,15023,"foo@v1 conflicts with bar@v2").
```

Listing 2: An example showing the specification of version conflicts in the format of Spack's concretizer. It would be derived from `foo`'s `package.py` file and specifies that `foo@v1` conflicts with `bar@v2`.

do not need to propagate error messages use standard ASP rules and constraints.

```
1   error(1, Msg) :- attr("node", Package),
2       conflict(Package, TriggerID, ConstraintID, Msg),
3       condition_holds(TriggerID),
4       condition_holds(ConstraintID),
5       not external(Package),
6       not attr("hash", Package, _).
```

Listing 3: An error rule from Spack's concretizer. This specifies that an error is present if a conflict exists and its conditions hold. This is how version conflicts are enforced in Spack's concretizer.

B. Probabilistic Answer Set Programming

While ASP provides a powerful interface for logic programming, it does not provide a mechanism for reasoning about uncertainty. Probabilistic Answer Set Programming is an extension of ASP that allows for the specification of probabilistic facts and rules. It makes use of the *stable model semantics* of ASP and extends it to probabilistic reasoning to find the *most probable* stable models. There are several proposed ways to extend ASP to support probabilistic reasoning [8]–[10]. Most of these center around replacing strict rules within stable model semantics with weighted rules that follow Markov Logic. This allows for probabilistic reasoning, stable model ranking, and recovering probabilities of facts and models.

Plingo [11], an extension of Clingo, implements several probabilistic variants of ASP. It extends the syntax of Clingo to allow for the specification of probabilistic facts and rules. These are then transformed into native Clingo rules and optimization statements to find the most probable stable models. Plingo also provides a mechanism for recovering the probabilities of individual facts and models. Listing 4 shows an example of a probabilistic ASP program that specifies the weight of particular facts using the *weight* rule. The example uses integer weights, but can also be modified to using floating point weights as log probabilities. Running the program in Plingo yields the stable models and their probabilities.

C. Predicting Build Probabilities

In order to obtain probabilities of build success for package configurations, we build upon the previous work BuildCheck [2]. BuildCheck is a graph neural network (GNN) based

```
1   bird(X) :- resident(X).
2   bird(X) :- migratory(X).
3   :- resident(X), migratory(X).
4   resident(jo) :- &weight(2).
5   migratory(jo) :- &weight(1).
```

Listing 4: An example of a probabilistic ASP program in Plingo from [8]. The *weight* syntax specifies a probability weight for a particular fact. This program can be read as follows. *X is a bird if it is a resident* (line 1). *X is a bird if it is migratory* (line 2). *X cannot be both resident and migratory* (line 3). *jo has been observed to be both resident and migratory with weights 2 and 1, respectively, higher being more trustworthy* (lines 4-5). This program will yield 3 stable models: $\{\}$ with probability 0.09, $\{resident(jo), bird(jo)\}$ with probability 0.67, and $\{migratory(jo), bird(jo)\}$ with probability 0.24.

tool that predicts the outcome of building a dependency graph. The underlying GNN model is based on a Graph Convolutional Network (GCN). The input to BuildCheck is a dependency graph where each node is a package and its features are an encoding of the package version. Edges in the graph represent dependency relationships in the build graph. The GNN model learns a representation of the graph and uses it to classify whether a package will build or not. A notable limitation of BuildCheck is that versions are encoded as one-hot vectors in the graph nodes, which means the model cannot be extended to new package versions without re-training from scratch.

BuildCheck was trained on a data set of over 40,000 Spack package builds. It was shown to predict 91% of builds correctly in its test set making it a perfect candidate for our studies. In order to use BuildCheck to predict build probabilities, we need to change the output of the model from classifying build success to regressing build probability. This is easily accomplished by removing the GNN's argmax computation after its softmax output layer, which can be interpreted as probabilities of build success. So in order to build our data set we create the dependency graph for a package configuration and run it through the modified BuildCheck model, which in turn outputs a probability of build success for the configuration.

III. Overview of Our Approach

In this section we provide an overview of our methodology for incorporating predicted build probabilities into Spack's version selection mechanism. The goal is to make use of historical build information to improve on hand-labeled version constraints and select package versions that are more likely to build successfully. Our approach focuses on incorporating predicted build probabilities into the logic program that Spack uses to select package versions. Figure 1 provides an overview of this approach.

First, we introduce a method for extrapolating build probabilities to new package versions. New package configurations are introduced rapidly in Spack and it is infeasible to collect new build data and re-train the GNN model every time package metadata changes. It is possible to re-train BuildCheck with

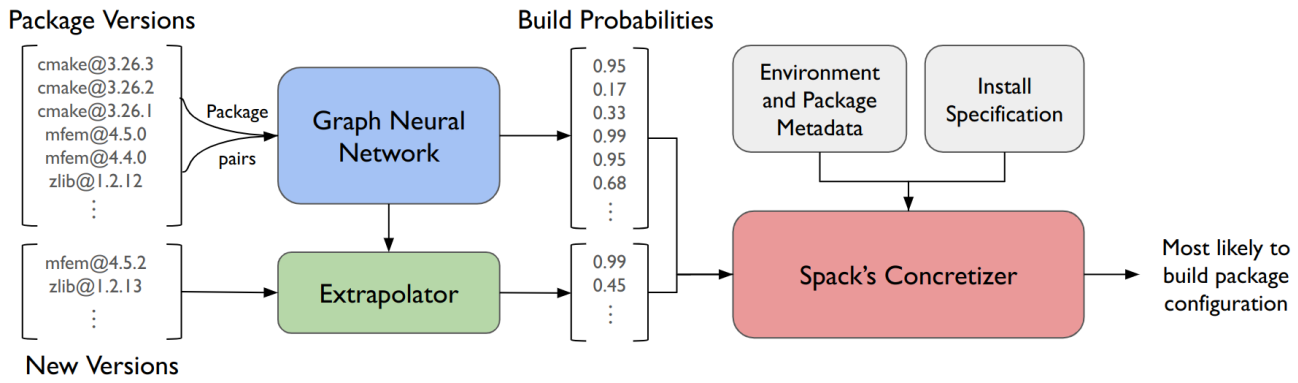


Fig. 1. Overview of our approach for incorporating predicted build probabilities into Spack's version selection mechanism. First, the graph neural network is used to predict the build probability for all pairs of packages. For new package versions, we extrapolate the build probabilities from existing package pairs. These build probabilities are then encoded into Spack's logic program alongside the existing metadata and install constraints. The modified logic program then selects the package versions that are most likely to build.

some frequency, but we need an approach to extrapolate build probabilities for short time horizons. We solve this by using expected probabilities for existing package pairs to extrapolate to new versions. When compared to several other extrapolation methods we find that this approach is the most accurate (see Section IV).

Now that we can extrapolate build probabilities to new package versions, we can use them to improve Spack's version selection mechanism. To accomplish this, our approach first computes the build probability for all pairs of packages. These pairs are then used to encode the build probabilities within Spack's logic program in order to select the most probable package versions. As finding good problem encodings for logic programs is non-trivial, we compare four different ways of encoding this information into the logic program using Answer Set Programming and its probabilistic variant Plingo (see Section V). These methods are compared over two different sets of packages in Spack and we report the build success rate for each method (see Section IX-A). Additionally, we compare the solve times for each of the new concretizer methods (see Section IX-B).

Using the best of these approaches we then investigate using probabilistic build data to improve Spack's existing hand-labeled constraints. This is accomplished by using query features in Plingo to get the probability of individual facts and entire stable models. These probabilities can be used to discover new likely constraints for the concretizer (see Section VI). We demonstrate the effectiveness of this approach by using it to improve the build success rate of the default concretizer (see Section IX-C).

IV. Extrapolating Build Probabilities for New Versions

> **RQ1** *How can we extrapolate build information to new package versions to mitigate frequent, expensive data collection?*

The existing BuildCheck tool provides us the ability to predict build probabilities over a set of known packages, however, due to its design it is unable to extrapolate to package versions that are not in its data set. The model is limited to known package versions due to its use of fixed sized vectors to encode package versions. This is limiting as package versions are added to Spack at a faster rate than new data can be collected and new models trained. Thus, a technique is needed to extrapolate build probabilities to the most recent few versions of a package.

We compare several methodologies for extrapolating build probabilities to new versions and compare their performance in this Section. We omit discussion of interpolating build probabilities between versions as it is unlikely for the collected data sets used to train BuildCheck to be missing values within version ranges.

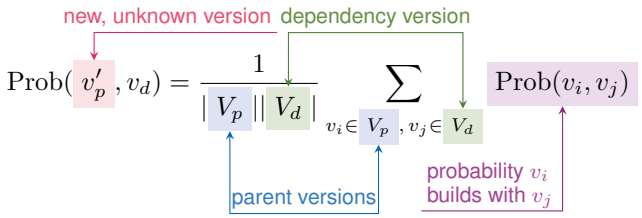
A. Details of Proposed Extrapolation Methods

We consider the problem of extrapolating build probabilities for a package-dependency pair to new versions of the package. More specifically, given a package p with a set of versions V_p and a dependency d with a set of versions V_d, we want to predict the build probability of p with d for a new version $v_p \notin V_p$. BuildCheck's GNN is not capable of this, however, we can use its output for known versions to make predictions for new versions.

Let $Prob(v_p, v_d)$ be the probability of building v_p with v_d. We want to predict this for an unknown v'_p (i.e., $v'_p \notin V_p$) using known $Prob(v_p, v_d)$ for $v_p \in V_p$ values. To estimate this we propose three different techniques: per-pair mean, nearest version, and regression. We try several variants of these methods and additionally compare to a baseline of guessing a constant value for each parent package. We describe each of these in detail below.

1) Per-Pair Mean: One of the simplest methods for extrapolating build probabilities is to take the mean of the known build probabilities for the package-dependency pair. For a

package p with versions V_p and a dependency d with versions V_d, this would be:



This can also be modified to take the mean for a fixed dependency version v_d instead of all dependency versions. This simpler average can be computed as $Prob(v'_p, v_d) = \frac{1}{|V_p|} \sum_{v_i \in V_p} Prob(v_i, v_d)$. These simpler extrapolations are easy to compute and reason with, however, they do not capture trends in versions over time or any anomalous behavior for a particular version.

2) Nearest Version: Another extrapolation method is to use the build probability from the *nearest* package-dependency pair. To accomplish this, we first need to define a distance metric between versions. This can be done by mapping packages into Euclidean space and using the Euclidean distance between versions.

Packages are mapped into $\mathbb{R}^P$ where P is the total number of packages in the data set. Each dimension corresponds to a particular package. The value of a package's dimension is an encoding of the package version. For packages with a standard versioning scheme (e.g., semantic versioning), we encode the version as a weighted sum of each component (major, minor, and patch version). In the absence of a standard versioning scheme, we encode the versions ordinally from 1 to n where n is the total number of versions for the package.

Once packages are mapped into $\mathbb{R}^P$, we can compute the Euclidean distance between two package versions. To extrapolate the build probability for a new version we find the nearest version in the data set and use its build probability. Once the data points are mapped into Euclidean space, this can be simply implemented using k-Nearest Neighbors (kNN) regression with $k = 1$.

3) Regression: The final extrapolation method we consider is to use a regression model to predict build probabilities. To accomplish this we transform the data set into a tabular form where each package version is represented in vector space. We use the same encoding as described in Section IV-A2 where each dimension corresponds to a package and the value is an encoding of the version. The vectors for the package and dependency are concatenated to form a single vector, which is then used as the input to a regression model that predicts the build probability for that package-dependency pair.

We consider several regression models including linear, AdaBoost, and XGBoost [12] regressors. We train each of these with an 80-20 train-test split using mean absolute error (MAE) as the training objective. Due to the large number of packages, and therefore features, we attempted to use principal component analysis (PCA) to reduce the dimensionality of the data set, but observed significantly worse performance.

B. Comparison of Different Extrapolation Methods

To compare each of these methods we evaluate them over the data set used to train BuildCheck. Let this data set be denoted as $\mathcal{D}$. We extract the build probabilities for each package-dependency pair, so that the final five columns of $\mathcal{D}$ are *package, package version, dependency, dependency version, build probability*. This data set has 62,075 rows and is further split into $\mathcal{D}_{\text{train}}$ and $\mathcal{D}_{\text{test}}$ with 20% of the pairs for each parent-child pair in the test set. To simulate extrapolating new packages on a short time horizon we limit the test set to only package pairs where one of the packages is at one of its three most recent versions. When evaluating the methods from Section IV-A we use $\mathcal{D}_{\text{train}}$ as the set of known package versions and $\mathcal{D}_{\text{test}}$ as the set of unknown package versions to extrapolate build probabilities for.

To compute the comparisons we implement each of the methods from Section IV-A in Python. The *Per-Pair Mean* method is implemented directly in Python and Numpy. The *Nearest Version* and *Regression* methods are implemented using the Scikit-Learn [13] package in Python. For the machine learning models we perform grid search over the hyperparameters to find the best possible MAE.

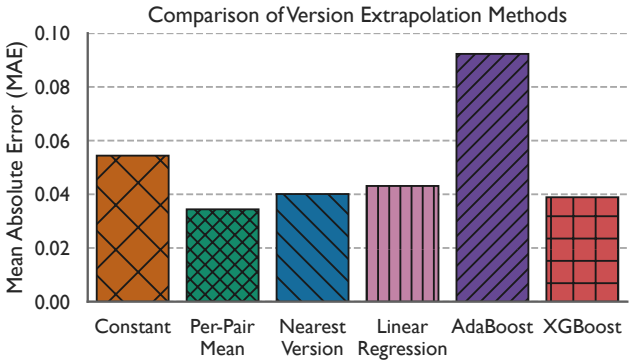


Fig. 2. Comparison of different methods for extrapolating build probabilities. The *Per-Pair Mean* method obtains the lowest MAE of 0.0344 when predicting build probability, and AdaBoost obtains the highest MAE.

Figure 2 compares the results for each method when evaluated over $\mathcal{D}_{\text{test}}$. We observe that the *Per-Pair Mean* method obtains the lowest MAE of 0.0344 when predicting build probability. This method is able to accurately predict the build probability within 0.0344 of the actual build probability. Thus, we can expect the build probabilities extrapolated from the model to be ± 3.44 percentage points of the actual build probability on average, which is a reasonable tolerance for our methodology. It is important to note that these approaches are simplifications and cannot model the complexities in the dependency graph as well as BuildCheck. However, they are useful for rapid extrapolation of build probabilities for new, recent package versions.

V. Incorporating Build Probabilities into Spack's Concretizer

Now that we have build probabilities readily available they need to be incorporated into the configuration selection mechanism of Spack. This is non-trivial as Spack's concretizer is implemented with logic programming and its strict rules do not readily allow for incorporating probabilities. Furthermore, the most difficult aspect of SAT problems and logic programming is generally finding correct and efficient *problem encodings* [1]. In this section we present four potential encodings for incorporating build probabilities into Spack's concretizer.

A. Encoding 1: Optimizing Version Selection in Clingo

The first approach modifies Spack's Clingo program directly by encoding build probabilities as integer weights and minimizing those weights during stable model selection. This approach loses the ability to recover the probabilities of individual models, but is the simplest to implement as it uses native Clingo. The approach is inspired by Spack's current version selection mechanism where versions are encoded 1 through N, where N is the total number of versions and 1 is the most recent version. The Clingo program is directed to minimize these version weights when selecting a stable model and, thus, more recent versions are favored in Spack.

We adopt this design to assign integers to package-dependency pairs based on build likelihood. Lower numbers are assigned to pairs that are more likely to build and higher numbers are assigned to pairs that are less likely to build. The Clingo program is then directed to minimize these weights when selecting a stable model. For example, consider that package *foo* depends on *bar*. *foo* at version *v1* can build with *bar* at version *v1*, *v2*, and *v3* with probabilities 0.9, 0.1, and 0.5, respectively. These would create *will_fail* facts in the program with weights 1, 3, and 2, respectively. The Clingo program is then directed to minimize these weights when selecting a stable model.

Listing 5 shows the modified Clingo program for the above problem encoding. Lines 2-4 demonstrate how build failure can be encoded as integer weights for package dependency pairs. Lines 6-7 use a choice rule (omitted for brevity) to include *pair_weight* facts in the stable model if that package-dependency pair is also in the stable model. Lines 10-14 then tell the Clingo solver to minimize the weights of the *pair_weight* facts in the final stable model. In practice, the build failure encodings (lines 2-4) would be generated offline and stored in a separate Clingo file. Only the choice rule and optimization criterion (lines 6-14) would be added to Spack's current concretizer program.

B. Encoding 2: Probabilistic Constraints

The second approach is similar to the first (Section V-A), however, we use Plingo to encode the build probabilities for the *will_fail* facts. This allows for the use of Plingo's capabilities for probabilistic reasoning and recovering the probabilities of individual facts and models. In this setup, the *will_fail* facts are encoded as probabilistic facts using the *weight* rule from Plingo. Section II-B describes *weight* rules in Plingo.

```
1  % fail probabilities for foo and bar
2  will_fail("foo", "v1", "bar", "v1", 1).
3  will_fail("foo", "v1", "bar", "v2", 3).
4  will_fail("foo", "v1", "bar", "v3", 2).
5
6  % pair_weight facts are included in the stable model if
7  % that package-dependency pair is in the stable model.
8
9  % optimize version pair weights
10 #minimize {
11     Weight@70+Priority,Package,Dependency
12     : pair_weight(Package, Dependency, Weight),
13     build_priority(Package, Priority)
14 }.
```

Listing 5: Example showing modification of Spack's Clingo program to weight version selection based on the "badness" of certain pairs. This is done by including *pair_weight* facts in the stable model if that version pair is also in the stable model. The Clingo solver is then directed to minimize the weights of these facts.

With the *will_fail* facts included in the program we then constrain against them. Constraints in Clingo are specified using `:-` statements. These are rules with an empty head and are read as "it is not the case that". The conjunction of facts after the `:-` cannot be included in the stable model. We use this to constrain against the *will_fail* facts when their particular package-dependency pair is also in the stable model.

Listing 6 shows an example of how this approach would be implemented. The probability encoding is demonstrated in lines 1-3. This would be repeated for all package-dependency pairs. Lines 6-9 then specify the constraint that would be added to Spack's current concretizer program.

```
1  will_fail("foo", "v1", "bar", "v1") :- &weight("0.1").
2  will_fail("foo", "v1", "bar", "v2") :- &weight("0.9").
3  will_fail("foo", "v1", "bar", "v3") :- &weight("0.5").
4
5  % constrain if will fail.
6  :-  depends_on(Package, Dependency),
7      attr("version", Package, Version1),
8      attr("version", Dependency, Version2),
9      will_fail(Package, Version1, Dependency, Version2).
```

Listing 6: Encoding build probabilities using Plingo's *weight* rule. This can be integrated with Spack's concretizer to select more probable versions for packages.

C. Encoding 3: Probabilistic Conflicts

The previous approach imposes a logical constraint on the final stable model with some probability based on the failure probabilities. This can be limiting in terms of the amount of information that can be propagated via constraints and how two or more constraints can interact. To address this we modify Spack's existing conflict specification mechanism to be probabilistic. This mechanism is described in Section II-A and Listings 2 and 3. Instead of creating *will_fail* facts and encoding them with build probabilities we instead introduce

conflict facts into the language (as in Listing 2) and assign them a probability. This introduces conflict statements into the program, which will produce errors if their conditions are met. Spack is in turn directed to minimize the number of errors in the final stable model.

An example probabilistic conflict fact is shown in Listing 7. The conditions for the conflict are specified in lines 1-8. These conditions are met when the packages and their specific versions are included in the final stable model. When these conditions are met the conflict (lines 9-10) is triggered. This approach is favorable to the other Plingo approaches in terms of implementation as it makes use of existing mechanisms within Spack's Clingo program.

```
1  condition(15022,"conflict trigger foo@v1").
2  condition_requirement(15022,"node","foo").
3  condition_requirement(15022,"node_version_satisfies",
4      "foo","v1").
5  condition(15023,"conflict constraint bar@v2").
6  condition_requirement(15023,"node","bar").
7  condition_requirement(15023,"node_version_satisfies",
8      "bar","v2").
9  conflict("foo",15022,15023,
10     "foo@v1 conflicts with bar@v2") :- &weight("0.9").
```

Listing 7: Example showing a probabilistic conflict fact for the conflict between *foo* at version *v1* and *bar* at version *v2*. These two conflict with probability 0.9. Each package and version is specified in a condition and condition requirements.

D. Encoding 4: Probabilistic Errors

The previous approach has the nice property that it cleanly integrates with Spack's existing conflict mechanism. However, it puts probabilistic constraints on the same optimization level as normal hard constraints, which may be less than ideal for final concretization performance. To introduce a new optimization level we create *build_error* facts based on *will_fail* facts and minimize them in the final stable model. Thus, we are increasing the probability of errors in a stable model when those models include package-dependency pairs that are likely to fail.

Listing 8 shows an example of this set up. As before, lines 1-3 encode the build probabilities for each package-dependency pair as *will_fail* facts using Plingo's *weight* rule. These are used in lines 5-12 to create *build_error* facts for each package-dependency pair if it is in the stable model. This rule is the only addition to Spack's current concretizer program.

VI. Using Causality to Identify New Conflicts

Incorporating historical build information into the configuration selection mechanism of Spack can increase the rate of successful package builds, but it does not eliminate the errors in package metadata that cause build failures. In this section we present a methodology for uncovering the causes of build failures using probabilistic build information. In order to accomplish this we can make use of the `query` feature in Plingo. This feature allows us to query the probability of individual facts

```
1  will_fail("foo", "v1", "bar", "v1") :- &weight("0.1").
2  will_fail("foo", "v1", "bar", "v2") :- &weight("0.9").
3  will_fail("foo", "v1", "bar", "v3") :- &weight("0.5").
4
5  build_error(50, Msg) :- attr("node", Package),
6      attr("node", Dependency),
7      depends_on(Package, Dependency),
8      attr("version", Package, Version1),
9      attr("version", Dependency, Version2),
10     will_fail(Package, Version1, Dependency, Version2),
11     not external(Package),
12     not attr("hash", Package, _).
```

Listing 8: Incorporating probabilistic errors into Spack's concretizer based on build probability. Probabilities are encoded using Plingo's *weight* rule. Then *build_error* facts are created for each package-dependency pair that is likely to fail.

(or atoms) being in the final stable model. For instance, `&query(attr("version", Package, Version))` will output the probability of a version for a particular package being in the final stable model. Another key Plingo feature we will use is the ability to list the probability of all stable models.

We first invert the existing Plingo concretizers in Section V to find the *least likely to build* configurations. This can be accomplished by changing all the `will_fail` facts into `will_build` facts with necessary adjustments made to the weights. We then sample a large number of stable models from the concretizer and set aside the least likely to build. We sample instead of using all stable models as the number of stable models is intractably large for most problems.

Within the least likely to build stable models we select the package pairs that have a `will_build` probability below some threshold α. If these package pairs are present in their stable models with a probability higher than β we consider them as new hard constraints. We then add these new conflicts to the existing Spack package metadata as shown in Listings 1 and 2. These new hard constraints supplement the existing hand labeled constraints within the package metadata and can be used to improve the concretizer.

To test this methodology we use the ECP-Proxy set of packages and the Plingo Prob. Conflicts concretizer to identify new hard constraints. We sample 10^6 stable models and use $\alpha = 0.05$ and $\beta = 0.9$ as thresholds for the package selection. We then add the new hard constraints to the existing package metadata and use the default Spack concretizer to build the packages. We compare the build rate of the Default, Prob. Conflicts, and Default + New Conflicts concretizers. This will show whether the new constraints are valid and if they improve the number of packages that build successfully.

VII. Implementing our Approach in Spack

The current Spack implementation does not support Plingo or custom concretizer implementations. This section provides an overview of how we modify Spack to test each of the new concretizers.

In order to implement and test each of the new concretizers described in Section V we need to modify Spack.

This can be accomplished by modifying the Clingo source code in Spack and the Python code that interacts with the Clingo API. In the Spack library all these source files reside in `spack/solver/`. The main concretizer program is in `concretize.lp` and the Python code that interfaces with the Clingo API is in `asp.py`. There are several other minor files related to the concretizer that we do not modify so they are not mentioned here.

We first incorporate the probability atoms into the solver. Each of the new concretizers in Section V relies on a list of probability atoms for each package-dependency version pair. We generate this file offline in each of the four formats using a Python script. The probabilities are stored in a separate file called `probs.lp`. These get included into the main solve call in `asp.py` when all the other external `.lp` files are loaded.

The next step is to modify the main concretizer program in `concretize.lp`. This is where we implement the main logic for each of the new concretizers. This is accomplished by directly changing the concretizer Clingo program. For the first concretizer approach (see Section V-A) this is sufficient, however, the other three concretizers require Plingo to be integrated. Fortunately, Plingo is implemented simply as an Abstract Syntax Tree (AST) transformation on top of Clingo's parser. This means that we can use the same Python code that interfaces with the Clingo API and apply the AST transformation to the concretizer program before solving. This AST transformation is applied directly after parsing the ASP program and right before the solve. The main effect of the AST transformation is mapping the soft rules from the Plingo program into optimization criterion in Clingo. The current Plingo implementation hard codes this to the highest optimization priority in Clingo, which interferes with Spack's existing optimization criteria, so we manually change the optimization level within Plingo's source. As an additional note we found that the current Plingo implementation does not work with Clingo `#heuristic` directives, so we disable them in Spack before calling Plingo. `#heuristic` directives are a way to provide hints to the solver about how to prioritize choices and are generally used to optimize solve times. We refer the reader to [11] for more details on Plingo's implementation.

VIII. Experimental Setup

With each of the problem encoding techniques implemented in Spack we can then test their effectiveness at improving the build success rate of package installations. This section provides an overview of how we test each of the new concretizers in Spack and the metrics we use to compare the results.

A. Setting up Build Experiments

To test each of the concretization algorithms we build a subset of packages from the Spack package repository using each concretization method: the E4S and ECP-Proxy application suites of packages. The first of these, E4S [14], is a collection of 80 unique packages that are used in the Exascale Computing Project (ECP) software stack. The latter set of packages, ECP-Proxy [15], is a collection of 22 packages that mimic the computational workload of larger, full scientific applications. Note that the GNN model from BuildCheck was trained on build data from E4S, but not ECP-Proxy.

To test the concretizers we build a set of 1000 packages comprised of the E4S and ECP-Proxy packages at their default settings and randomly selected versions. To randomly select versions we sample a subset of dependencies and then randomly request versions for those dependencies. Each build is done with each concretizer method implemented in Spack. We concretize and build each package separately (Spack is able to concretize environments, or sets of packages, all at once) and provide the `--fresh` concretize option to keep Spack from aggressively reusing existing binaries from local or remote caches. Additionally, we test all the packages for each concretization method together and then clear the Spack environment before testing the next concretizer to prevent any build from being reused. All of these builds are done on the Quartz cluster at Lawrence Livermore National Laboratory. It has 3018 nodes, each with an Intel Xeon E5-2695 v4 processor, 36 cores, and 128 GB of memory. The builds are run as an array of Slurm [16] jobs with each job running on a single node and using 32 cores to build in parallel.

B. Metrics Used to Compare Encodings

To compare the results of each concretizer we use two metrics: the ratio of successful to total builds and the number of packages that fail with an experimental concretizer but succeed with the default concretizer. The first metric is the most important because it directly measures the effectiveness of each concretizer. This metric can lie between 0 and 1 with 1 being a perfect build success rate. We define this as shown in Equation 1.

indicator function
1 if the package built successfully, 0 otherwise

$$\text{build\_success\_rate}(P_i) = \frac{1}{|P_i|} \sum_{p \in P_i} \mathbf{1}\{p \text{ builds}\} \quad (1)$$

set of package specifications to install

While we are most concerned with the build success rate, we also want to know whether it is introducing any new build errors. This could happen despite an overall improvement in the build success rate and is, thus, not captured by the previous metric. To measure this we compute the ratio of new package build failures to total package build failures. This metric also has values lying between 0 and 1, however, here 0 is a perfect score indicating that no packages that built previously fail with a new concretizer. We define this as shown in Equation 2.

concretizer being tested

default Spack concretizer

$$\text{new\_failures}(P_i, C_j) = \frac{\sum_{p \in P_i} \mathbf{1}\{p \text{ builds with } C_{\text{default}}, \text{ not } C_j\}}{\sum_{p \in P_i} \mathbf{1}\{p \text{ build fails with } C_j\}} \quad (2)$$

Both Equations 1 and 2 provide a summary of how a particular concretizer improves the number of build successes for package installs. Being scalar values they can be compared directly across the different concretizer implementations to determine which is the most effective. We also record the solve time of each new concretizer, however, the evaluation of this metric is limited due to the bug in Plingo preventing `#heuristic` directives from being used (see Section VIII), which are a major optimization within Spack's solver.

IX. Results

In this section we present the results for the different concretizer implementations and the causality study.

A. Comparison of Build Success Rates

> **RQ2** *How can probabilistic build information be used to select better dependency versions and improve the likelihood of successfully building a package?*

Figure 3 presents the build success rates for each concretizer on the ECP-Proxy and E4S package sets. For both sets of packages the new concretizers outperform Spack's default concretizer by up to 13 percentage points. For E4S all new concretizers perform the same, while for ECP-Proxy the new concretizers vary. This is likely due to the fact that the Build-Check model is trained on data from the E4S set of packages, but not ECP-Proxy. Predicted probabilities are likelier to be noisier for the ECP-Proxy packages which could exacerbate the differences between the proposed concretization methods.

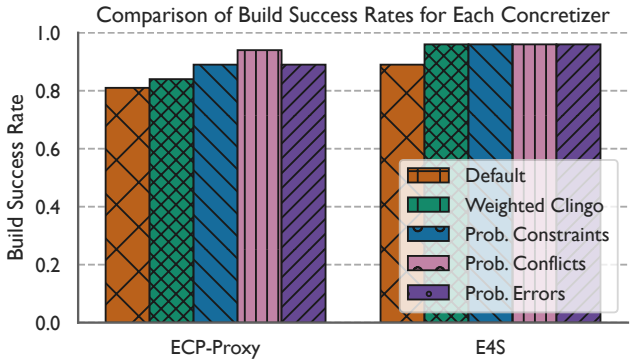


Fig. 3. Build success rates for each concretizer on the ECP-Proxy and E4S package sets (higher is better.) For both sets of packages, the new concretizers outperform the default concretizer. For E4S, all new concretizers perform the same, while for ECP-Proxy, the probabilistic conflicts in Plingo perform the best.

For the E4S set of packages there is a 7 percentage point improvement in build success rate for all new concretizers. The build success rate improvement ranges between 3 and 13 percentage points for the ECP-Proxy set of packages. Using the probabilistic conflicts in Plingo results in the highest build success rate for ECP-Proxy. The probabilistic conflicts in Plingo (see Section V-C) encode information directly into Spack's existing version conflict mechanism. This allows the concretizer to use the information in a way that is more consistent with the existing concretizer.

We observe that new_failures(P_i, C_j) is 0 for all concretizers C_j. None of them introduce new failed builds for any package. This shows a strict improvement over the default concretizer without hurting any existing results. This is critical for the adoption of the new concretizers as they can be integrated without any risk of breaking existing workflows and creating a bad user experience.

Despite an improvement in build success rate there are still packages that concretize, but fail to build. This is due to build failures originating from issues other than version selection. Packages can fail to build due to compiler mismatch, missing dependencies, invalid build flags, etc. For instance, the version of Spack used for testing, 0.20.1, does not treat compilers as dependencies, so their versions are handled separately within the concretizer. We noticed that the failure cases for the four proposed concretizer methods in our testing were all either due to compiler and/or build flag mismatches between dependencies.

B. Comparison of Concretization Times

Figure 4 presents the concretization times for each concretizer over the entire package set. The two native Clingo concretizers, Default and Weighted Clingo, both take around the same amount of time to solve. They are also both much faster than the Plingo concretizers and have less variance in their solve times. This could be due to the fact that Plingo is built on top of Clingo and adds additional overhead.

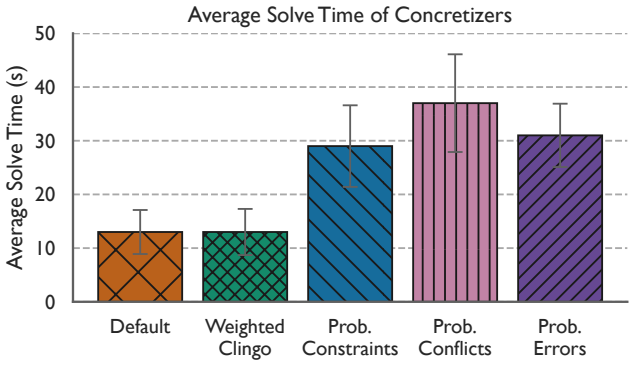


Fig. 4. Concretization times for each concretizer over the entire package set (lower is better.) The new concretizers are slower than the default concretizer. However, Plingo currently does not support the same optimizations that Spack's concretizer uses.

These results mostly shed light on how ideal the Weighted Clingo solution is. It improves on the number of packages built (see Section IX-A), has little overhead, and is easy to implement. The slowness of the Plingo concretizers is expected as they add more complexity to the solve call by adding new minimization criteria (see Section II-B). More importantly, the `#heuristic` directives that Spack uses for optimization do not currently work in Plingo and likely

contribute significantly to the observed slowdown in Figure 4. Furthermore, the duration of a solve call will be heavily dependent on the set of packages, current environment (other existing installations), problem encoding, and hardware.

C. Understanding Causality of Build Failures

> **RQ3** *Can probabilistic version selection mechanisms be effectively used to determine causality for failed builds?*

Figure 5 presents the results of using Plingo to identify new hard constraints for the concretizer. The results are presented over the ECP-Proxy set of packages. We observe that the new conflict statements, or version constraints, improve the default concretizer by 3 percentage points. This shows that we are able to glean build failure causality from the Plingo-based concretizer.

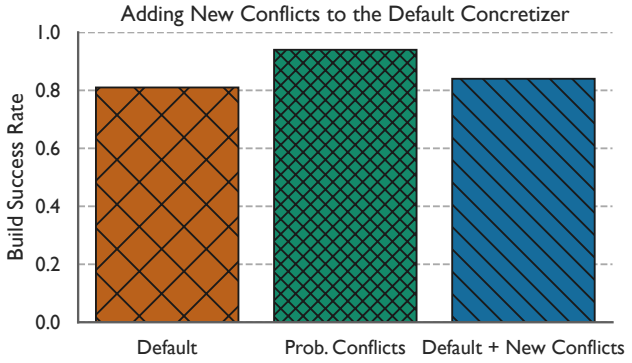


Fig. 5. Adding new conflicts to the existing Spack concretizer based on the outputs of the Plingo concretizer yields an improvement of 3 percentage points (for ECP-Proxy packages.)

However, the performance is not as good as using Plingo directly to label conflicts with probabilities. This is expected as the new hard constraints are only for packages we are certain will not build. The probabilistic concretizer has more information about the build success rate of each package and can therefore make better decisions. This motivates the use of the probabilistic concretizer in addition to any other technique for hard constraint discovery as we can improve the number of valid configurations by using both.

X. Limitations

Our methods for predicting build probability and selecting build configurations are limited to reducing compile time errors and not version conflicts errors that arise only during dynamic linking or run time. For instance, Spack has over 2200 Python packages available for which most would not create errors during install if there was a version conflict. These errors would appear during run time and may even only be triggered by specific run time call paths. Run time errors caused by version mismatches could be incorporated into our methodology by also considering post-install tests for packages. However, this requires that the package has a test suite and that the test suite has proper coverage of the package's functionality. This type of testing is not currently available for most of the packages relevant to this study preventing us from including it in our methodology.

XI. Related Work

As package managers have grown more complex they have needed strong solutions to the problems of dependency management and package versioning. Since version compatibility is NP-complete, many rely on SAT solvers or other variations to select versions [17]–[19]. These solvers rely on existing package metadata, such as versions and constraints, for inputs into their solver. These metadata are provided by package authors and kept up to date by package managers. However, as discussed in this work, these metadata can be wrong or incomplete due to the complexity of managing a large set of fast changing packages. Due to this problem some works have looked at new version selection policies to improve over existing methods.

One such work [20] proposes *Wisdom of the Crowd* for version selection. This method selects the most popular and highly used version of a package for the dependency version. While this can improve over existing hand selection methods, it can still introduce errors. In particular, it will not be able to prevent version conflicts that only manifest in uncommon and unpopular library features. Other simpler policies typically rely on Semantic Versioning [21] to select versions. This involves matching compatible sub-versions of a package to determine if they are compatible [22], [23]. These simpler approaches have been popular as they are easy for developers to reason through and implement. However, they are not able to capture the complexity of dependency versioning and can lead to errors, particularly when overly relied upon.

To build on methods such as *Wisdom of the Crowd* and Semantic Versioning, several works have developed more complicated version selection policies that use binary analysis or historical build information [24]–[32]. Menon et al. [24] propose a Bayesian Optimization based strategy to find the best version of a package to install that improves over *Wisdom of the Crowd*. Other works [25] look at the binary compatibility of packages to determine if they are compatible. Xu et al. [26] propose a methodology for editing the source code to prevent version conflicts. Many of these approaches focus on identifying version conflicts and/or fixing them. However, they all differ from our work in that they do not integrate build likelihood from historical build data into an existing build system and package manager to improve the likelihood of successfully building a package.

XII. Conclusion and Future Work

We have demonstrated a methodology for integrating probabilistic build information into a package manager's configuration selection system. Using this methodology, we have implemented a prototype system that is shown to improve the package manager's successful build rate by up to 13

percentage points. Additionally, we have demonstrated how to extend our methodology to new package versions as they are added to the package manager. The methodologies presented in this paper can be extended to other build and package management tools that implement version selection semantics. In future work we plan to extend this methodology to more pieces of package meta-data such as compiler and build flags. We also plan to further investigate how to easily transfer build probabilities to new systems and architectures. Additionally, we would like to extend the concretizers to optimize build outcomes other than success and failure, such as build time, memory usage, and performance of the final binary.

ACKNOWLEDGMENT

This work was performed in part under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-857730). This material is based upon work supported in part by the National Science Foundation under Grant No. 2047120.

REFERENCES

[1] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*, 1st ed. Addison-Wesley Professional, 2015.

[2] H. Menon, D. Nichols, A. Bhatele, and T. Gamblin, "Learning to predict and improve build successes in package ecosystems," in *International Conference on Mining Software Repositories*, ser. MSR '24, 2024.

[3] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral, "The spack package manager: bringing order to hpc software chaos," in *SC15: International Conference for High-Performance Computing, Networking, Storage and Analysis*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2015. [Online]. Available: https://doi.ieeecomputersociety.org/10.1145/2807591.2807623

[4] T. Gamblin, M. Culpo, G. Becker, and S. Shudler, "Using Answer Set Programming for HPC Dependency Solving," in *Supercomputing 2022 (SC'22)*, Dallas, Texas, November 13-18 2022, lLNL-CONF-839332.

[5] M. Gebser, R. Kaminski, B. Kaufmann, and T. Schaub, *Answer Set Solving in Practice*. Springer International Publishing, 2013. [Online]. Available: https://doi.org/10.1007/978-3-031-01561-8

[6] V. W. Marek, I. Niemelä, and M. Truszczynski, "Origins of answer-set programming - some background and two personal accounts," *CoRR*, vol. abs/1108.3281, 2011. [Online]. Available: http://arxiv.org/abs/1108.3281

[7] M. Gebser, B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub, and M. Schneider, "Potassco: The potsdam answer set solving collection," *AI Communications*, vol. 24, no. 2, pp. 107–124, 2011.

[8] J. Lee and Y. Wang, "Weighted rules under the stable model semantics," in *Proceedings of the Fifteenth International Conference on Principles of Knowledge Representation and Reasoning*, ser. KR'16. AAAI Press, 2016, p. 145–154.

[9] C. Baral, M. Gelfond, and N. Rushton, "Probabilistic reasoning with answer sets," *Theory Pract. Log. Program.*, vol. 9, no. 1, p. 57–144, jan 2009. [Online]. Available: https://doi.org/10.1017/S1471068408003645

[10] L. De Raedt, A. Kimmig, and H. Toivonen, "Problog: A probabilistic prolog and its application in link discovery," in *Proceedings of the 20th International Joint Conference on Artifical Intelligence*, ser. IJCAI'07. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007, p. 2468–2473.

[11] S. Hahn, T. Janhunen, R. Kaminski, J. Romero, N. Rühling, and T. Schaub, "Plingo: A system for probabilistic reasoning in clingo based on LP^{MLN}," in *Rules and Reasoning*, G. Governatori and A.-Y. Turhan, Eds. Cham: Springer International Publishing, 2022, pp. 54–62.

[12] T. Chen and C. Guestrin, "Xgboost: A scalable tree boosting system," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[14] "The extreme-scale scientific software stack," https://e4s-project.github.io/index.html, accessed: 2023-09-30.

[15] "Ecp proxy applications," https://proxyapps.exascaleproject.org/, accessed: 2023-09-30.

[16] "Slurm workload manager," 2020. [Online]. Available: https://slurm.schedmd.com/documentation.html

[17] R. Di Cosmo, "EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies," INRIA, Tech. Rep., May 15 2005, hal-00697463.

[18] F. Mancinelli, J. Boender, R. di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*, 2006, pp. 199–208.

[19] Python Software Foundation, "New pip resolver to roll out this year," Online, March 23 2020, https://pyfound.blogspot.com/2020/03/new-pip-resolver-to-roll-out-this-year.html.

[20] Y. M. Mileva, V. Dallmeier, M. Burger, and A. Zeller, "Mining trends of library usage," in *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, ser. IWPSE-Evol '09. New York, NY, USA: Association for Computing Machinery, 2009, p. 57–62. [Online]. Available: https://doi.org/10.1145/1595808.1595821

[21] T. Preston-Werner, "Semantic versioning 2.0. 0," 2013.

[22] A. Decan and T. Mens, "What do package dependencies tell us about semantic versioning?" *IEEE Transactions on Software Engineering*, 2019.

[23] J. Dietrich, D. Pearce, J. Stringer, A. Tahir, and K. Blincoe, "Dependency versioning in the wild," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 349–359.

[24] H. Menon, K. Parasyris, T. Scogland, and T. Gamblin, "Searching for high-fidelity builds using active learning," in *Proceedings of the 19th International Conference on Mining Software Repositories*, ser. MSR '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 179–190. [Online]. Available: https://doi.org/10.1145/3524842.3528464

[25] B. E. Cossette and R. J. Walker, "Seeking the ground truth: a retroactive study on the evolution and migration of software libraries," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, 2012, pp. 1–11.

[26] S. Xu, Z. Dong, and N. Meng, "Meditor: inference and application of api migration edits," in *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 2019, pp. 335–346.

[27] P. T. Nguyen, J. Di Rocco, R. Rubei, C. Di Sipio, and D. Di Ruscio, "Recommending third-party library updates with lstm neural networks," 2021.

[28] Q. He, B. Li, F. Chen, J. Grundy, X. Xia, and Y. Yang, "Diversified third-party library prediction for mobile app development," *IEEE Transactions on Software Engineering*, 2020.

[29] P. T. Nguyen, J. Di Rocco, and D. Di Ruscio, "Mining software repositories to support oss developers: A recommender systems approach." in *IIR*, 2018.

[30] Z. Sun, Y. Liu, Z. Cheng, C. Yang, and P. Che, "Req2lib: A semantic neural model for software library recommendation," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2020, pp. 542–546.

[31] P. T. Nguyen, J. Di Rocco, D. Di Ruscio, and M. Di Penta, "Crossrec: Supporting software developers by recommending third-party libraries," *Journal of Systems and Software*, vol. 161, p. 110460, 2020.

[32] A. Ouni, R. G. Kula, M. Kessentini, T. Ishio, D. M. German, and K. Inoue, "Search-based software library recommendation using multi-objective optimization," *Information and Software Technology*, vol. 83, pp. 55–75, 2017.