MASTER THESIS

Bc. Peter Fačko

Package Manager for C++

Department of Software Engineering

Supervisor of the master thesis: RNDr. David Bednárek, Ph.D.
Study programme: Software Systems

Prague 2024

I would like to thank my supervisor, RNDr. David Bednárek, Ph.D., for his time and valuable advice during the work on my thesis.

I would also like to thank my family and friends for their support during my studies.

Title: Package Manager for C++

Author: Bc. Peter Fačko

Department: Department of Software Engineering

Supervisor: RNDr. David Bednárek, Ph.D., Department of Software Engineering

Abstract: C++ currently lacks a standard package manager. While two projects competing for dominance exist, the most popular dependency management method is still to manage the packages manually. Although some progress has been made towards establishing a shared solution, most of the projects aimed at resolving the issue have resulted in ecosystem fragmentation. To address this issue, we developed a package meta-manager to expand on the progress while avoiding extra fragmentation. The project unifies the best features of multiple managers by integrating them under one interface. We define the subset of the project dealing with dependency resolution as a formal model. We also employ a full SAT solver to explore a solution for complete dependency resolution.

Keywords: package manager, C++

Názov práce: Package manager pre C++

Autor: Bc. Peter Fačko

Katedra: Katedra softwarového inženýrství

Vedúcí diplomovej práce: RNDr. David Bednárek, Ph.D., Katedra softwarového inženýrství

Abstrakt: V C++ aktuálne chýba štandardný package manager. Zatiaľ čo existujú dva projekty, ktoré súperia o prevahu, najpopulárnejším spôsobom spravovania balíčkov ostáva manuálne spravovanie. Hoci nejaký posun smerom k jednotnému riešeniu nastal, väčšina projektov s cieľom vyriešiť tento problém mala za následok roztrieštenie ekosystému. Ako reakciu na daný problém sme vytvorili package meta-manager, ktorý stavia na doterajšom pokroku a vyhýba sa ďalšej fragmentácii. Projekt zjednocuje tie najlepšie schopnosti viacerých managerov pomocou ich integrácie do jedného rozhrania. Časť projektu, ktorá sa zaoberá riešením závislostí, sme zadefinovali aj ako formálny model. Taktiež pre preskúmanie možností úplného riešenia závislostí zapájame v projekte SAT solving.

Kľúčové slová: package manager, C++

Contents

Introduction

Nowadays, software is commonly distributed in the form of packages, which are archives containing everything needed to install a piece of software. To help automate package handling, package managers were developed. The primary purpose of package managers is to provide a consistent way of installing software [1].

Package managers can be approximately split into two categories: *dependency* managers and *system* package managers. A dependency manager is a program developers use to manage a project's dependencies. They are often language-specific because the projects and all their dependencies frequently use only a single programming language. Some well-known examples are C++'s Conan, Python's pip, JavaScript's npm, and Rust's Cargo.

A system package manager is a program responsible for managing installed programs in an operating system distribution. Although it is not their primary purpose, they are often used also as dependency managers. This type of usage is prevalent in the C++ community. Examples of this type of package manager include archlinux's pacman, Debian's apt, Gentoo's Portage, and Google's Play Store for Android.

More formally stated, *package management*—the principal focus of package managers—is the process of resolving requirements into a set of packages and subsequent installation of those packages. It also typically involves further management of those installed packages, such as upgrades and removal.

Requirements passed to package managers can take almost any form but are usually short strings with a simple format. Their purpose is to express a constraint on the resulting package set. An example from the pacman package manager would be a simple `python` for requiring that the Python package is present in the result. Managers usually allow specifying multiple requirements at once, in which case the requirements form a logical conjunction of the constraints they represent.

A *package* is, in the most general meaning, a combination of two parts: the files forming the software the package is distributing and the metadata consumed by the manager to handle the package successfully. Software in compiled languages has two forms: the source code and the binary. The term "package" could also refer to both forms or just either of the two. From a package manager's point of view, the term could also be synonymous with a whole software project. We will define the exact terminology we use in the context of this thesis when we start designing our project.

Packages themselves can also impose requirements on the package set. A specific type of such a requirement is a *dependency*. `python` is not only a requirement but also a package that has dependencies. Dependencies are conditional requirements that must hold only if the dependent package is present in the package set. `python` depends on `openssl`, so any package set containing `python` must also contain `openssl`.

Installation means making the package available for consumption in a particular context. For example, installation can mean copying the package's files into a directory. Another installation method would be creating references to the package files, allowing indirect consumption through the build system.

Packages are typically collected in *repositories* to simplify consumption. The user can configure their manager to use a set of repositories. The set can usually have an order of priority associated with it to resolve package name conflicts. Conflicts can happen when two repositories contain a package with the same name. Repositories can be maintained with a specific purpose; for example, Linux distributions maintain repositories of packages that essentially define their distribution. They are not simple collections of packages but also ensure compatibility and stability guarantees.

Another name for resolving requirements is *dependency resolution*. All package managers offer *correct* dependency resolution, which means that each installation produced by the manager satisfies the requirements from user input. However, the resolution in some managers is *incomplete*, meaning it can fail to find a solution even if one exists. Other managers don't allow arbitrary version requirements but only accept those that can be resolved completely quickly. Moreover, system managers typically don't provide multiple versions of the same package, reducing their resolution complexity. Overall, a typical approach is to avoid a dependency resolution that is both potent and complete.

In the context of compiled languages, such as C++, there are two application interfaces: source (API) and binary (ABI). The ABI is very important from the point of view of package managers, as its incompatibility can cause severe issues for package consumers. Examples of properties influencing a binary's interface are the API, the compiler version, and which flags are passed to the compilation. There are also other influences on the ABI. Therefore, to be more precise, the ABI is the product of the API and some properties of the build context. When a package manager stores binaries in a cache, it has to be able to determine their equivalence. This type of equivalence is mainly determined by ABI equivalence.

Although we believe developing a generic package manager is better than a language-specific one, we decided to focus on C++ for multiple reasons. One is that it is a language lacking a good package management solution, which makes it an interesting area to explore. Another reason is that we consider C++ to be such a complex language that solving package management in its context would probably cover a large region of use cases for many other languages. Lastly, it is simpler to focus only on a limited domain.

We also decided to focus only on the Linux environment, as it best suits the needs for developing this type of software.

The State of Package Management in C++

The Standard C++ Foundation surveys C++ developers annually. The survey contains these two questions: "Which of these do you find frustrating about C++ development?" and "How do you manage your C++ 1st and 3rd party libraries?". For at least three years in a row, in 2022 [2], 2023 [3], and 2024 [4], the top answers for both questions were the same. "Managing libraries my application depends on" frustrates the users the most, and most users said: "The library source code is part of my build". Here is an example selection of languages with a standard or a de facto standard package manager: D, Go, JavaScript, .NET, Python, Ruby, and Rust. These two facts show that 1) a package manager is a desired and common feature of any programming language, and 2) C++ does not have a package manager, and it is its most frustrating non-feature.

In this state of the ecosystem, developers employ three different methods of managing their packages. Two are fundamentally ill-suited for the purpose they are used for: manual management and using system package managers. The third is dependency managers, which are a good choice in general. However, we assess the state of the ecosystem as fragmented because there have been multiple dependency managers developed for C++, some with a decent adoption, and yet, the surveys show that their usage is still not nearly as widespread as in other languages, where a package manager is standard [5].

Manual Management

The most direct and, in some ways, most straightforward method of managing packages is manual management. Using this method, the developer manually installs all of their project's requirements. This sometimes involves also building the dependencies.

One problem with this approach is reproducibility. By reproducibility, we mean ensuring that the software consumer will use the same dependencies with compatible versions. The project might depend on a library the developer is unaware they have installed on their system. When dependencies are packaged with the project, this can be detected if the developer tests the packaging. However, manual instructions might have errors revealed only after the distribution. The developer can partially solve this problem with manual management by installing dependencies into a special directory used only for the project. Although this does not solve hidden dependencies on system libraries, using this method will discover regular dependencies during the development.

Another issue lies with package distribution. When the project needs to be distributed (packaged), the developer has two choices: package the dependencies with the project or provide instructions to install dependencies manually. Both of these approaches are not ideal. Distributing the libraries is often not possible due to their licensing. On the other hand, manual installation would require the package consumer to build and install a library that they have no motivation nor skill to use.

System Package Managers

A better approach to package management is to utilize system package managers. As package managers, they automate the process of building and installing packages.

Because of this, distribution is simple; the user only needs a list of packages to install with their system manager.

However, the issue of reproducibility is even worse than with manual management, as the host system is the only context from which the project's dependencies are consumed. Container technology, which combines well with system package managers, provides a solution. Containers allow the creation of a separate system used only for the project's build, which can thus contain only the necessary dependencies. Unfortunately, containers cannot be used in all contexts, as they share some components with the host system—most notably the kernel.

Another problem with system managers is that their advantages are leveraged only when the user uses the distribution's official repositories. Using unofficial repositories is more complicated and may present an obstacle to the user. Official repositories are also not without a problem. They have a stricter policy for package inclusion, and because of that, the project's required packages might not be present in the repositories.

Dependency Managers

Although not as popular as they could be, dependency managers have recently gained significant popularity in the C++ community. The two most used are Conan and vcpkg. They can handle dependency management completely.

Reproducibility can be achieved by installing the packages into a special directory and only allowing consumption of declared dependencies. This can be achieved by setting the correct variables during the build process.

Problems arise with distribution. Dependency managers handle consuming libraries by project developers well. However, distribution to the end users is left unmanaged, with only helper tools provided. The end user is not expected to use a dependency manager.

Package Manager Security

Package managers are essential to a system's security because of their privileged position in managing installed software. A *dependency confusion attack* is a type of cyberattack where the attacker suddenly causes package manager users to download malicious software instead of a regular package [6]. This attack exploits a feature of some package managers where a repository from which a package is fetched is dynamically evaluated based on repository contents.

Consider the most straightforward situation where the manager configuration contains two repositories, R_1 and R_2, in that order of priority. Suppose there is a known package p in R_2 and not in R_1 that the user wants to install. They can request p, and everything will work as expected. However, when an attacker wants to make the user install their package, they only need to upload p into R_1. Since the location from which p is fetched is determined as the first repository that contains a package with such name, p is fetched from R_1 during the following invocation of the manager after the attacker uploads the malicious package.

Some package managers have implemented solutions that prevent these attacks, but most have not addressed this issue.

Goals

Based on the research above, we conclude the following goals for the thesis:

- Integrate multiple package managers used in the C++ ecosystem so that packages presented to the user are the union of all packages the integrated managers provide. This integrating application will be called the *meta-manager* and constitutes the main software part of this thesis.

- Design the project's architecture as loosely coupled so it supports easy expansion of integrated managers.
- Support complete dependency resolution with version requirement formats inherited from the integrated managers.
- Implement a mechanism guarding against dependency confusion attacks.
- Support building packages from source. This feature will be accompanied by binary caching and a mechanism for communicating the ABI of binaries.
- Allow installing packages into any directory. The resulting directory shall be usable as a container image root file system. In particular, the resulting images should be usable as build contexts for building packages.
- Allow the meta-manager to be executed in a container.
- To allow for rigorous design correctness checking, construct a formal model generalizing a subset of package manager functionality required to satisfy these goals.

Thesis Overview

First, we analyze specific solutions offered by current package managers related to our work in chapter 1.

In chapter 2, we design the architecture and structure of our application.

With the design completed, chapter 3 formalizes a subset of the design concerning dependency resolution.

Chapter 4 explains the most important choices and challenges with implementing the system.

The application implemented in this thesis serves primarily as a proof of concept. The Conclusion discusses significant shortcomings and possible improvements.

The thesis attachment is discussed in Appendix A.

1 Analysis

In this chapter, we will analyze several package managers that are used in the C++ ecosystem. We will explore the most essential features supplied by the managers and the particularities in their implementation. This analysis will serve as the basis for designing our project. Two managers, pacman and Conan, will be analyzed in more detail, as they will be the managers integrated into the meta-manager. The information gathered from the analysis of other managers will be compiled as a set of design patterns.

1.1 pacman

pacman is the default package manager of the archlinux Linux distribution. It is, therefore, a system package manager. Its modern versions are designed as a front-end to the libalpm library. By pacman, we will refer to the whole application as experienced by an end-user. We will use libalpm when referring specifically to the application's library part.

pacman manages binary packages. It understands a package as an archive of application files and metadata about the package relevant to pacman. Packages are installed into a root directory, and files from the archive are unpacked relative to that root. This model suits distributing binary packages well because it only considers the final form of the files, which only need to be copied.

This analysis is based on the archlinux wiki pages, and pacman's and other related software's manpages [7].

1.1.1 Package Metadata

Each package states its metadata, which is the information the package manager uses to handle the package. Its primary purpose is to identify the package and provide information about package relationships.

Name

Each package has a name—a simple string by which other packages and the user refer to it. The name is one of the package's interfaces.

Version

Packages state their version to communicate API compatibility. The pacman version is the source code version and should correspond to the version released by the software author. Versions are also simple strings, except they cannot contain hyphens.

Release

As the package version should correspond to the source code version, changes to package metadata must be reflected elsewhere. The release value serves this purpose. It is usually an integer but can be any string. A typical reason for incrementing the release is a bug fix in the packaging script.

pacman often refers to package version as a concatenation of version and release, so typically, a user would see a package identified as, e.g., `python 3.12.4-1`, where `3.12.4` is the version and `1` is the release.

Provide

Each package can provide any number of *virtual packages* or simply*provides*. A provide specification consists of a string and an optional version specification. Provides are interfaces the package offers in addition to its name.

Examples of provided virtual packages are package `bash` providing `sh` and `openssl` providing `libcrypto.so=3-64`.

Dependency

The most basic type of package relationship is a dependency. Its format is a string followed by an optional version requirement specification. The version specification can be exact or a comparison. The possible values are: `p<v`, `p<=v`, `p=v`, `p>=v` and `p>v` with the obvious semantics. The pacman program suite provides the `vercmp` command as an implementation of the comparison logic. Similarly, libalpm provides an exported function with the same utility.

When a package in an installation depends on `p`, a package with the name `p` or a package providing `p` must also be present in the installation. When a package depends on `p@v`, the package providing that interface must additionally provide it at a version satisfying comparison `@v`. Note that packages can provide virtual packages without a version; such a package cannot satisfy a dependency containing a version requirement, as the version comparison is not defined in that case.

Examples of dependencies are `zlib` depending on `glibc`, `curl` depending on `libz.so=1-64`, which is provided by `zlib`, and `glibc` depending on `linux-api-headers>=4.10`.

Conflict

A conflict is also a relationship between packages. Its format is identical to the format of a dependency. The semantics are inverted, so when a package in an installation conflicts with an interface, that interface must *not* be provided in that installation. Version comparisons apply the same way as with dependencies—only the providers of the interface at a satisfactory version are considered.

Examples of conflicts are `jack2` and `pipewire-jack` providing and conflicting with the interface `jack`, as `pipewire-jack` is a replacement of `jack2`. Since they both conflict with a virtual package, a hypothetical third replacement would need to provide the same interface and would conflict with both without requiring any change in the metadata of other packages.

1.1.2 Installation

pacman is designed to manage system packages in a particular directory. All package requirements apply to the context of one directory. A typical installation directory is the root directory of the managed system. However, pacman also allows installation into an arbitrary directory. In one directory, there can be only one installed version per package name, as the installed packages are assumed to be global for a whole system.

A package upgrade is in pacman a package removal and a subsequent installation of a package with the same name but with a different version. To support package upgrades, pacman keeps a database of all installed packages with their respective file paths in the root directory. This way, an upgrade can be done by removing the old version and copying the new one. The files that need to be removed are looked up in the database. pacman calls this database the *local database*.

pacman installs and removes packages using transactions. If one package fails to install, all other changes in the same transaction are reverted. This is achieved with the help of the local database.

In the metadata, each package can specify hooks to be executed before or after a transaction. The hook specified to run post-transaction only runs when the transaction commits successfully. Hooks can also be installed in a package-independent location. These hooks can specify conditions of execution relative to multiple packages. The hooks are simple commands with arguments and optional standard input containing information about why the hook was executed. An example would be executing `useradd` after the installation to ensure a user is present in the system. An issue is that the hooks must be executed in a chroot environment as they expect to be run relative to the root directory. This means some commands might not execute equivalently to a regular global root.

1.1.3 Repositories

For pacman to be useful, it also allows the user to download packages and interact with package *repositories*. A remote package repository is a collection of packages. Each repository can have multiple servers that serve the repository's data. These servers are called *mirrors*.

The user configures pacman to use any number of repositories by specifying a repository name with a list of mirrors to be used to fetch packages from that repository. Together, these repositories are unified from the point of view of pacman, and packages are fetched from any repository from the list. When two repositories provide a package with the same name, the first listed repository with that package overrides all others.

To reduce the bandwidth required to use pacman, all repository data except the actual package archives is fetched before it is used. This way, pacman does not need to communicate through the network when resolving user inputs. The database where the repository data is stored is called the *sync database*. Its location is typically in the installation root directory but can be specified with an option as an arbitrary directory.

Each repository contains only one version per package. With the same rule applying for an installation root, pacman is suited for a *rolling release* model. This means that a repository (or rather a union of repositories) is a sort of authority over which versions are the best, and an installation directory can deviate from this authority. A system upgrade is then an upgrade of all packages in the installation directory with a different version from the repository.

pacman does not support *partial upgrades*, which are upgrades to only a subset of packages deviating in version from the repository. In other words, the only supported sets of package versions are those that have, at some point, resided in a repository. The reason for this is that the repositories mainly exist as an authority over package compatibility. When a user installs package versions that have not been tested against each other, their compatibility cannot be ensured by the distribution maintainers.

When a user requests a package to be installed, the default behavior is to resolve and install its dependencies as well. This resolution is simplified because each package only provides a single version. Virtual packages introduce choice into the resolution, but pacman solves this by prompting the user to select the desired provider.

pacman provides a package cache where all package archives are downloaded into and installed from. The required packages are looked up in the cache when dependency resolution is done. When the cache is missing the packages, they are downloaded from locations specified by the repositories. The cache location has its default value but can be defined as an arbitrary directory.

1.2 The Arch build system and the AUR

The *Arch build system* (ABS) is a system for building packages from source code suited for the archlinux distribution. ABS package metadata is defined using a file called PKGBUILD. This file is a Bash script containing package metadata and instructions for building.

Each package can list its build dependencies, which are packages required to be present when building the package. Normal dependencies are also required during the build, so the difference between the two is that build dependencies must be present *only* during the build. An implicit build dependency for all packages is the `base-devel` package.

This analysis is based on the archlinux wiki and makepkg's manpages.

1.2.1 Building Packages

To build a package, PKGBUILD is read by the command tool makepkg. In most circumstances, executing `makepkg` in a directory containing a PKGBUILD is sufficient to build the package.

To ensure the correctness of stated dependencies, the build is expected to be run in a minimal environment called a *clean chroot*. This means that only the explicit and transitive dependencies are supposed to be present in the environment in which `makepkg` is executed.

makepkg cannot be run as the root user. This is because PKGBUILD is a shell script and can therefore contain arbitrary commands. The archlinux wiki page suggests running the command as the nobody user when no regular user is available.

The behavior of makepkg can be configured to use different compilation flags. This allows users to specify options tailored to their specific system and achieve some performance gains. Another option is to use the compilation flags for a custom build of standard packages, i.e., to produce SELinux enabled binaries.

Package source code is expected to be downloaded during the build process. This is typically done by cloning a source control repository. The source control program is required to be specified as a build dependency. Therefore, a typical build dependency is the `git` package. Another option would be downloading a source archive using package `curl`.

1.2.2 The Repository

The *Arch User Repository* (AUR) is a collection of user-provided package build descriptions. The descriptions mainly consist of PKGBUILDs. Packages in the AUR are unsupported by the archlinux distribution and designed to be wholly user-maintained.

One consequence of the unsupported status of the AUR is that the distribution does not maintain package upgrades, and the user needs to rebuild packages manually when a new version is released. This is because the PKGBUILD can be written in a way that the downloaded source's version depends on the time at which the package is built; a so-called "live at head" approach is a typical example of this.

The packages in the AUR can be fetched from two locations. Both location's URLs begin with `https://aur.archlinux.org`. The primary distribution method is a separate Git repository for each package located at `/package.git`. A secondary location is snapshot archives at `/cgit/aur.git/snapshot/aur-hash.tar.gz`. The `hash` value in the URL corresponds to the commit hash.

An issue with a system where all versions of one package reside in a single Git repository is the translation of versions to commits. The `pkgctl` utility replaces forbidden characters in the version string to produce a tag. Although simple, this method is not guaranteed to work because package maintainers are not required to tag commits with version strings. The only sure method is browsing the whole history and looking for the most recent commit with the specific version listed in the PKGBUILD.

The AUR provides two interfaces for fetching package metadata. One is an RPC interface over HTTP returning JSON. Users can search packages by name and query a particular package's metadata through the HTTP interface. The interface has a maximum request size and limits the rate at which requests can be made. The second interface is a simple file with metadata for all packages written in the JSON format. Both interfaces provide package metadata only for the most recent versions.

1.3 Conan

Conan is an open-source C/C++ dependency manager. In 2023, Conan 2.0 was released, and we will analyze only this version as it is a backwards-incompatible major release.

Conan is written in Python and can be used directly as a library. It also provides an API that is currently (2024) in its early stages and has been flagged as an experimental feature.

This analysis is based on the official Conan documentation available online [8].

1.3.1 Package Metadata

In Conan, the user specifies package metadata in a single file called conanfile. Two formats are supported: a simple declarative text file or a Python module. The Python format is functionally a superset of the declarative one, so we will focus only on Python conanfiles. The conanfile and all other files needed to handle a package created by the package maintainer are called a *recipe*. The recipe, therefore, excludes the source files.

Each conanfile specifies a version of the package which should correspond to the source version. When package metadata is somehow changed with the source version the same, the information needs to be stored somewhere. Therefore, Conan always calculates the hash of a recipe and uses that hash as part of its version. The hash is called recipe *revision*, and it uniquely identifies a recipe.

The most important metadata specified in a conanfile is the requirements. A requirement has analogous semantics to a pacman dependency. The requirement format consists of a package name and a version expression. The version expression can either be a specific version or an expression with comparison and logical operators. The fact that a Conan requirement specifies directly a package name is in contrast with pacman dependencies which specify interfaces. Conan has no concept of virtual packages.

The advantage of specifying package metadata as a Python module is that the metadata can be parametrized and calculated on demand. The possible parameters are options and settings. The difference between the two is that options are specified per package while settings are used for each package during one Conan invocation. The settings are typically used for `Debug` vs `Release` mode or for changing the target architecture. Options can, for example, specify whether the specific package should be built as a static or shared library. Together, options and settings can be aggregated into files called *profiles*. When Conan processes a conanfile, it executes the Python methods which can read the passed options and settings and return different values based on those parameters. An example would be an optional feature of a package that requires an additional dependency.

1.3.2 Repositories

Conan allows consuming packages when developing a project as well as creating and maintaining them. To distribute packages, Conan provides the concept of repositories, similar to pacman. The repositories contain recipes that are fetched by the Conan client.

Since Conan does not keep any local database of available recipes, it always needs to download information through the network. Because of this, it does not implement complete dependency resolution and performs only a simple greedy algorithm, which can fail even if a correct solution exists. There are multiple issues on GitHub complaining about the resolution not finding an obvious solution.

1.3.3 Binary Packages

Since Conan is suited for C and C++, it can handle building from source files. To support this, Conan also allows the user to specify build requirements. In a cross-compilation context, these packages require a different architecture and possibly also other settings, so Conan allows specifying a different profile for the build and target context.

The built packages are tagged with a *package ID*. It is typically a hash of the settings, options, recipe, and dependencies, although its calculation can be customized by the package maintainer. It uniquely identifies a package binary and, therefore, allows the binaries to be stored in the repositories. When Conan needs a binary and calculates the package ID, if a package with the same ID is already present in the repository, it can safely download and use it. Conan therefore also supports distributing binaries.

To allow packages to communicate information to depending packages, Conan provides a feature called *package info*. The info has the format of a Python object and can contain any JSON serializable data. Each built package provides one package info. A good example of package info usage is passing the compiler flags used to build the package, a well-known feature provided in a different context by pkg-config.

Many projects use complicated build systems that need additional data generated for a successful build. Conan provides the mechanism of generators, which are simple scripts that generate helper files based on a dependency tree. A typical example is a CMake generator that puts the package's version and dependencies in a `CMakeLists.txt` file.

Conan uses extensive caching. All downloaded recipes and built packages are cached, while recipe revisions and package IDs serve as cache keys. When consuming packages, Conan is designed to be able to provide the package without any copying by pointing the build to the location in the cache using generators. Because of this, a Conan installation just fetches the package and its dependencies into the cache and executes the specified generators. Conan also allows the user to export recipes and built packages from the cache and import them into a different cache location.

1.4 Package Manager Design Patterns

We analyzed more package managers from the C++ ecosystem and one popular manager from another language and detected several common design patterns. The analyzed package managers except the two already covered are vcpkg, Nix, tipi.build, build2, and npm.

To help the understanding of the derived patterns, we describe them using a fixed structure similar to the one used in the article "A pattern language of an exploratory programming workspace" [9]. The structure will be the following:

First, we introduce the problem context for the pattern. Then, we describe the problem solution, which is the pattern description. Those are the parts defining the pattern. Next comes a section with pattern drawbacks, as patterns don't typically come without a price. Since these patterns are derived from a specific package manager analysis, we next mention the particular examples of pattern occurrence in actual software. Lastly, we reference related patterns, as the discovered patterns can be grouped by their problem scope into a more understandable structure.

1.4.1 Package Context

Problem

Managed packages are always inherently relative to some context. In the typical case of a system package manager, the context is the whole system. The fact that the context in such cases is vast and implicit makes it very difficult for the manager and the user to take advantage of package contexts.

Consider a situation where two projects, project A and B, are developed on the same machine with package management done by a typical system package manager. Project A requires package p with version v_1, and project B requires p with version v_2. Suppose the simultaneous presence of versions v_1 and v_2 would lead to the diamond problem. Since the context for the system manager is the whole system, the manager cannot allow both versions of p to be installed.

Moreover, when upgrading, removing, or changing the installed packages, there is a slight possibility that some binary from the package will run during the transformation process. Having this operation be non-atomic could lead to an unexpected runtime behavior of the package.

Solution

To make the package context concept more useful, managers can make package contexts explicit. That can be done in two ways. One is documentation, where package contexts are explicitly discussed as a property of the manager. Another is implementing package contexts as an entity in the manager's software architecture. An explicit entity could allow the user to create, configure, and reference contexts, enabling additional features. To address the broadness typical for managers not concerned with package contexts, the manager provides a way to specify the affected area for a package context or requires the package consumer to define the context to which the consumption is relative.

A manager providing explicit contexts allows an easy solution to multiple projects requiring different package versions because each project can be relative to a different context. A way to avoid offering arbitrary contexts while solving this problem is to provide project-local contexts, where each project created by the manager gets assigned a new context associated only with the new project.

Atomic upgrades, rollbacks, and similar features can be implemented using explicit package contexts. For each transformation, the manager can create a new context with the transformation applied to it and leave the old context intact. When atomic switching between the used context is applied, e.g., changing a symbolic link, the atomic transformation can be done by the following. First, a non-atomic transformation is performed in the new context, which isn't yet used by anything and doesn't affect anything. Then, the manager atomically switches the context.

Drawbacks

As the concept of a package context is always present in the package manager design, introducing explicit contexts has no functional drawback. However, since having the contexts explicit increases the complexity of a manager's design, there is a drawback of increased development time.

Also, contexts are not optional when implemented explicitly regarding the user's choice of utilized features. This can lead to a steeper learning curve for the user, which could also be considered a disadvantage over a more implicit solution.

Known uses

vcpkg and npm provide explicitly documented project-local contexts.

build2 provides explicit context creation with linking, which allows the user to create a base context with shared content used by multiple inheriting contexts.

Nix provides user-local packages and atomic package transformations using package contexts and symbolic links. The steep learning curve is most evident in this manager.

1.4.2 The Manager as a Package

Problem

A package manager is software, so it needs to be distributed to its end users. We assume software distribution is complex, so practices that reduce that complexity can often be worth the development time. The manager also needs to be maintained as software. This mainly means being updated and installed or removed.

Software distribution is simultaneously one of the main features of package managers. However, the distribution capabilities provided by the manager can be limited, as it typically only needs to support distributing packages for the specific environment it is tailored to.

Solution

The solution to the problem of package manager distribution is to provide the manager as a package managed by the manager. This significantly reduces the complexity of its distribution since the design of package distribution implemented by the manager is reused and reapplied for this purpose, and all its guarantees are carried over.

As a byproduct, the manager can then be installed, updated, removed, and generally reliably maintained in a well-understood way by users and developers.

Having the manager provide itself as a package requires the manager to be implemented to handle its own installation. The installation could also involve building, which would need to be handled by the manager.

A problem with this pattern is the cyclical situation where a user wants to install the manager, yet the manager is required for its installation. This must be solved by a bootstrap method, where the manager provides another way of distribution. This distribution can be minimal as it only needs to support installing the manager itself.

As a bonus, this pattern provides the manager with a good tutorial for the users, as the first required step of using the manager is to install it. The developers can then control the user's first experience because they are in control of the manager, and no other package is involved in this beginning stage yet.

Drawbacks

The main problem with this pattern is that the package manager must be implemented with the feature in mind. This might have small implications for some managers, but it can significantly constrain the implementation for managers that distribute only from source in a specific language.

It also isn't clear that every manager would benefit from this pattern, as it depends on the package management features provided. Some managers might have such a feature set that this pattern introduces complexity with minimal benefit.

Known uses

npm provides itself as a package, although the usage is a bit more complicated, and the typical usage requires an additional "meta" manager.

build2 provides itself and multiple other components of its toolchain as packages. The manager is well integrated into its ecosystem by supporting build-time dependencies.

1.4.3 Default Values

Problem

A package manager is complex software that requires the user to input a significant amount of configuration. Although the documentation for the options, parameters, and switches is typically provided, users can still find it challenging to understand the functionality of all the configuration parameters offered. Some parameters don't have to be specified, so if the user doesn't understand them, they can leave them out. However, with some parameters, the user must pick one option from many, which is a problem if they don't understand the implications of individual values. The steep learning curve could deter users from using complex and feature-rich managers.

Solution

The solution is to use default values for most of the configuration. If the manager provides sensible default values for the newly introduced parameters, it can expand its feature set without deterring new users.

One sensible choice for a value is the value used by most users or use cases. Another option for the default is to choose the safest option, which might not be the most common one but is the least probable to cause problems to the user.

Drawbacks

The Default Values pattern is best manifested when a user doesn't understand a particular option but doesn't have to because the manager uses a default value with a high chance of working correctly without the user noticing. This advantage is also a significant drawback as the option is, by design, hidden from the user, thus possibly skewing the user's understanding of the manager. It is often the case that a configuration option is tightly associated with a specific feature. This means that the only place where a user might find out about a defaulted option and, thus, commonly also the related feature is the documentation. Unfortunately, the documentation is often not consulted unless it is obvious and necessary. Default values usually make the consultation of documentation necessary but not obviously so.

Known uses

Almost all software utilizes default values.

Related patterns

A more advanced and complex pattern in the area of default behavior is the Scanning pattern.

1.4.4 Scanning

Problem

The Default Values pattern applies only when a configuration parameter accepts a finite set of values or a number. For example, if the parameter type is a string, developers often find it challenging to choose a sensible value that applies to most cases.

Solution

A good solution for some parameters with complicated types is not to provide a default value and throw an error when the user doesn't specify some value explicitly. However, a lot of configuration passed to the manager often repeats the same values that are part of the package's content. These are, for example, the package name, version, binary name, and so on.

The manager uses a scanning tool that parses the package content and looks for the desired values. This might be an external tool for source code or a parsing library for a known format such as JSON.

Package managers often come with their so-called manifest files containing the package metadata. Still, this pattern is more concerned with scanning for values in a package's content that are not explicitly for manager consumption.

Sometimes, the user wants to use a value for an option related to the value found inside the package content, but not exactly as found. In this case, the manager can provide another parameter that specifies the transformation that ought to be done to the found value.

Drawbacks

Scanning is more complex and involved than just picking one default value, so it makes the manager harder to use.

The other difficulty is in the manager's implementation complexity, as scanning might require a lot of additional development. For example, parsing a directory structure is quite simple, but parsing code or arbitrary text files for default values is much more difficult.

Known uses

tipi.build uses scanning as part of its "build by convention" design.

Related patterns

Default Values deals with a simpler case of default configuration.

1.4.5 Lockfiles

Problem

Many managers allow users to specify a dependency with a version range or without specifying a version requirement at all. This means that the manager might have multiple package versions to choose from.

A widespread behavior is to choose the newest available version. Although any backward-compatible version is supposed to work correctly, users sometimes wish to achieve reproducible builds. Reproducible builds ensure greater portability and improve debugging by disallowing a dependency to change the project's behavior. Reproducibility may also help prevent dependency confusion attacks.

Solution

A solution for reproducible builds is to use a so-called *lockfile*. A lockfile contains information about the built package's dependency tree. This information is complete, meaning the whole dependency tree can be reconstructed using the lockfile.

It is common for managers to implement lockfiles as literal files that are passed as arguments when invoking the manager.

Drawbacks

An obvious drawback is that lockfiles require an additional structure in the project configuration, which must be passed to the manager during the invocation and complicates the manager's usage. The lockfile location can be set to a default value, making the feature less discoverable.

Some could view the solution of lockfiles as a symptom of a bigger problem: a manager choosing the newest version of a package by default.

Known uses

Conan and npm offer lockfiles as one of their main features concerning versioning.

Related patterns

Baseline is a more robust solution to the problem of an updated package dependency. However, it comes with more constraints.

1.4.6 Baseline

Problem

One approach to package version specification is for the manager to allow arbitrary version ranges and pick the newest available version. In that case, a problem of unreproducible builds arises, which can be solved by lockfiles.

From another point of view, the possibility of a new version breaking its dependents is itself a problem. Lockfiles can also be considered a problem, as they are a whole new mechanism developers must develop and users learn to use.

Another problem related to package versions is that a newer version of a package can cause a dependency error requiring an additional fix. That means depending only on package versions when considering compatibility is not a reliable and complete solution.

Solution

When a package manager allows specifying only left-bounded version ranges while always choosing the minimal version satisfying all dependencies, the problem of a new package dependency version breaking the project is solved [10].

Package managers often provide global package repositories. These can be utilized to address the problem of versioning unreliably, ensuring compatibility. A way to ensure maximal compatibility between packages in a registry is to have a CI system in the repository that triggers with any change, builds all affected packages (dependencies and dependents), and rejects any change to the repository which causes a build failure. For each package upgrade, a new snapshot of the repository is registered. These snapshots are called baselines. The user then specifies dependencies to the manager relative to some baseline, and the manager resolves dependency versions to at least the value in the baseline. This way, the manager provides a mechanism for ensuring package compatibility with the option to use newer, untested versions of packages. Although the newer versions might still break compatibility, the user must explicitly specify them.

Drawbacks

Although the Lockfiles pattern arguably introduces unnecessary complexity to a manager's design, Baseline comes with its complexity and is also more involved.

Moreover, in contrast to lockfiles, usage of baselines is not optional as they must stand at the base of dependency version specification in managers using them.

Known uses

From the analyzed packages, only vcpkg offers baselines.

Related patterns

The Lockfiles pattern solves a similar problem more directly.

1.4.7 Binary Cache

Problem

Package manager developers must make an important design decision: distributing source code or binary. These types of distribution are not mutually exclusive, but each has its incompatible advantages.

Distributing source code is highly portable because each platform should be able to build binaries for itself. It might also result in faster code as additional optimizations specific to the target architecture could be applied. Lastly, there is no need to ensure binary compatibility when distributing source code, as no binary is distributed.

On the other hand, binary distribution is faster because it doesn't involve build time, which is especially slow in C++. Build times are the second most common frustrating feature reported by C++ developers in the annual survey done by the Standard C++ Foundation. Moreover, not involving a package manager in package building simplifies its design.

Solution

The manager can provide binary caching to achieve the benefits of both building from the source and directly distributing binaries. When a user requests a package, the manager first looks up the package in the cache, and only if it is missing there will it invoke the build from the source.

Determining whether the cache contains some package is not trivial and requires a complex solution that ensures binary compatibility. However, the manager can always ignore the cache and build from the source, as caching is only an optimization, and building always yields the correct binary.

Drawbacks

This pattern has problems to solve, but no substantial drawback exists. Caches are an optimization and can be ignored.

Known uses

vcpkg, Conan, Nix, and tipi.build all provide binary caching. The only manager building from source not supporting binary caching is build2, but it, too, has plans to support it in the future.

2 Design

All package managers share the same basic operational flow. They all accept requirements from the user, resolve those requirements into a set of packages with exact versions, then fetch products of those packages and install those products into a particular context.

The specific managers' semantics will be implemented as modules separate from the main application. The meta-manager will, therefore, define several interfaces through which the modules will be called. Even though they are separate from the software point of view, we will develop modules to integrate pacman, the AUR, and Conan as part of this thesis.

Before we start designing the application, it is useful to introduce solid terminology. From the point of view of the meta-manager, *package* refers to the specific packaged source code with metadata. By *package group*, we will mean packages from the same sequence of versions. So, in the context of pacman, `python` is a package group and `python 3.11.5-1`, `python 3.11.5-2`, and `python 3.11.6-1` are all separate packages. Note that the number after the dash is a release number in pacman and is not part of the proper version of the package. However, it changes package metadata, and therefore, we consider two pacman packages with a different release number as separate packages. A collection of files produced by a build is often called a binary. But since it is not strictly necessary for the contents to be complied binaries, we will refer to this collection as a *product*. In pacman, `python 3.11.5-1` would also refer to a product since it distributes binary packages. In Conan, `zlib` is a package group. `zlib/1.3.1#f52e...` is a package because revisions change metadata. And `zlib/1.3.1#f52e...:c810...` is a binary built from that package and is, therefore, a product. We only used the first four characters from the hash values for brevity.

Conan does an extra step at the end of the invocation and produces so-called generators. They depend on the products and a set of generator names. The output is a directory. We could consider generators a part of the installation step, but we would unnecessarily introduce more coupling. Generators can be created in a location independent of the installation context.

Since one goal is to produce image root directories, our interface accepts requirements and generator names as input and returns two outputs: the generator directory and the image root directory.

We can use the same primary phases all managers have, and we end up with a structure described in Figure 2.1. The following sections will explain the structure in more detail, starting from the bottom of the diagram and going up.

2.1 Installation

To design a common interface for installation, we will recapitulate what we know about the specific managers we will integrate.

pacman installs archives into a directory. The directory is intended to be used as a root of a system. The installation of multiple packages happens as a transaction. Before and after a transaction, installation hooks can be executed. Hooks are arbitrary commands run in the context of the installation directory.

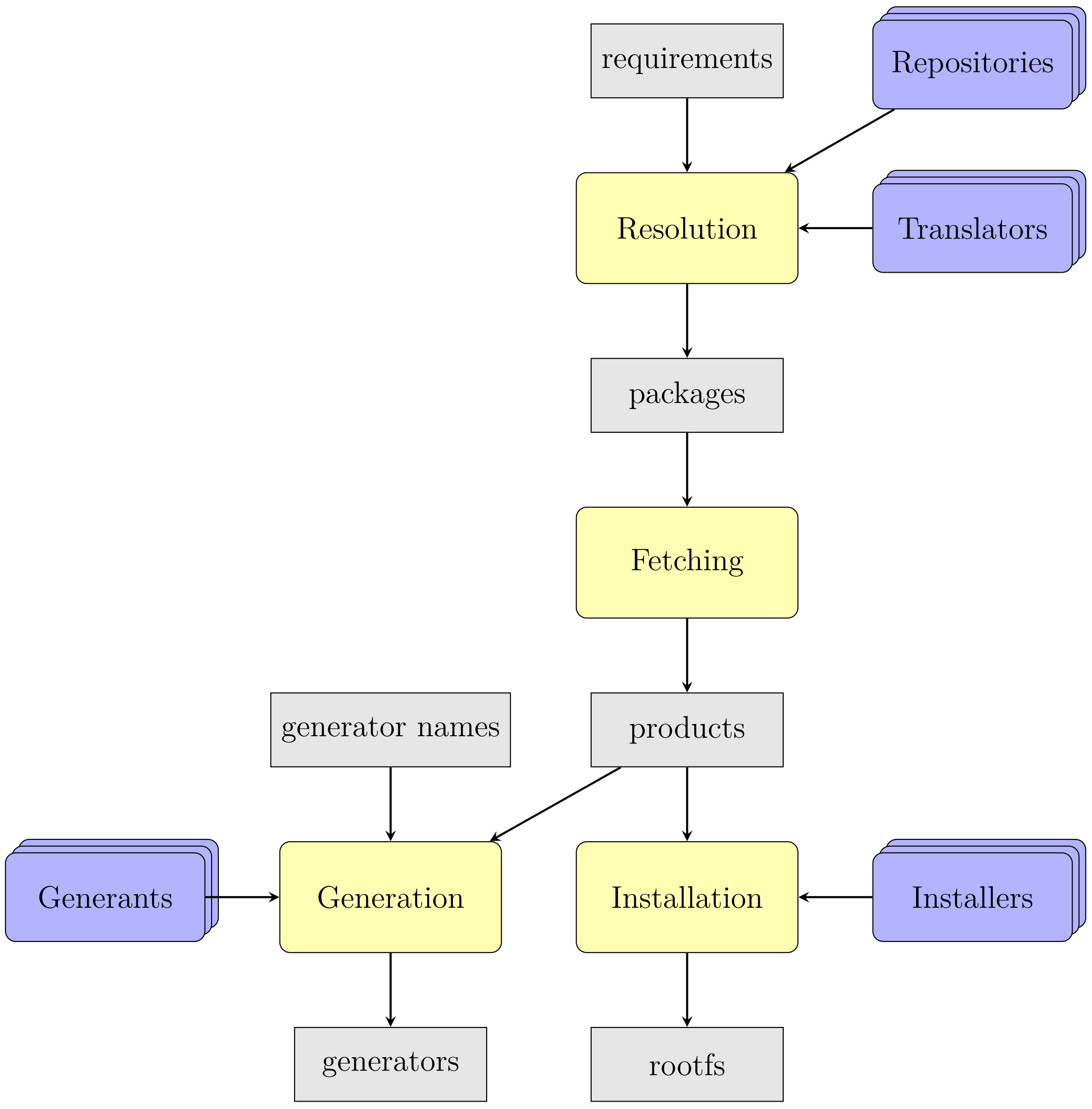

Figure 2.1 Structure of the meta-manager

Consuming Conan packages happens indirectly by setting the correct variables to the paths in Conan's cache. Installation in this manager, therefore, only makes the desired product available in the cache.

We know that we need to produce a container image. pacman installation can be used directly, as the resulting directory is already intended to be used that way. So, we must put the Conan cache into the directory. The only way to do this is to put it into a predefined location. A natural choice is the default location. The default location of Conan cache is the folder `.conan2/` in the current user's home directory. Containers are typically run as the root user mapped to the current user on the host. So, we can use `/root/.conan2/`. A disadvantage of this choice is that users other than the root cannot use the directory. An alternative would be to use a directory such as `/usr/share/conan-cache/`. A disadvantage here is that a command running in such an image would have to set the `CONAN_HOME` environmental variable. For now, we decided on the simpler option of the default cache location in the root's home directory.

To separate the logic of specific managers, we will use different installation implementations as modules. Modules used to install a product will be called *installers*. An installer will accept two inputs: the product's location as a filesystem path and the destination directory. It will return nothing. The destination directory will have the product installed when the module's invocation is finished.

Therefore, product installation has two parameters: the product and the installer. We will allow the user to configure a mapping of different installer names (strings) to specific installers. The fetching of a product will then need to provide the product and the name of the installer by which the product ought to be installed. By allowing the build process only to specify a simple string, we will enable the user to customize the installer behavior by switching to a different installer configured under the same name.

2.2 Generation

Conan accepts a set of generator names and products and creates a directory with the generator contents. The interplay between generators is not defined, and it does not matter in what order the generators are invoked. Therefore, invoking Conan once with a set of generators is equivalent to invoking it multiple times with one generator at a time.

pacman does not make use of any generation, so we can define the interface according to Conan and call it a *generant*. A generant accepts *one* generator name, a set of product paths, and a destination path. It returns nothing and produces the generator's content in the destination path based on the product set.

Meta-manager inputs contain generator names, but the meta-manager needs to know which generator to execute for each name. Therefore, the user must configure a mapping of generator names to generants. We should allow regular expressions here because a single generant typically handles many different generator names. A regular expression is a good solution if all names have the same prefix.

Note that Conan uses the term "generator" for multiple different entities. In our terminology, there will be a `conan` generant, implementing the logic for *all* Conan generators. That is the reason why a generant accepts a generator name. We refer to Conan generators such as `CMakeDeps` as generator names. By generators in the context of the meta-manager, we refer to the directory of files resulting from the phase called generation.

2.3 Fetching

pacman fetches already-built packages by downloading them as archives. Returning a URL from which an HTTP GET request would download a package would be a sufficient interface for supporting pacman.

Conan is more complex because it builds packages. It first calculates the package ID and tries to look up the corresponding product in the repository. If the repository does not contain the product, the build must follow. The build happens on the host machine in a context with different environmental variables. By setting the correct variables, Conan can, for example, use a cross-compiler and a different version of the build system. Build dependencies are installed into the cache and pointed to as needed.

2.3.1 Product Interface Information

To support binary caching, we need a mechanism for capturing product equivalence. Two equivalent products can be used interchangeably in an installation. If we determine the equivalence class and already have a product from that class in the cache, we can use it and avoid building it.

We will use a concept called *product interface information* (PII). The PII will contain the ABI information the depending packages need to build correctly. The PII needs to be calculated, and one of the inputs of this calculation will be the PIIs of the package's dependencies. As we already covered, the ABI also depends on properties of the build context, and therefore, there needs to be a second argument, *build context interface information* (BCII). Each build context will have to provide a BCII, which will be passed on to the calculation of the PII. A repository should govern the calculation of a PII for a specific package, as it governs all package metadata.

We can simplify using the product interface information as the product ID and binary cache key. If the product should communicate some other information downstream that doesn't affect the product interface, the package maintainer can still include it in the product's files.

2.3.2 Dependency Ordering

When calculating PII, the PIIs of the dependencies are the inputs to the calculation. SAT solving does not return a dependency graph, only a model—a set of packages. So, we need another mechanism for constructing the graph: interfaces and interface dependencies. Interfaces will be simple identifiers (strings). A package p_1 depends on p_2 iff there is an interface I that p_1 depends on and p_2 provides. Using this semantics, we can construct a graph. An error is raised when the graph contains a cycle, so we can assume it is acyclic—a DAG.

2.3.3 Building

To build deterministically, we will build in containers. We need an image, a working directory, an environment, and a command to execute inside the image. We can then specify inputs and outputs as either standard IO or paths where files are to be expected.

One option to specify an image is to define a tag or a Dockerfile.

Another is to invoke the meta-manager recursively and create the image that way. In that case, the specification would be in the same format as input to the meta-manager: requirements and generator names.

Packages from the AUR are expected to be built in a system where both the runtime and build dependencies are present. This is not optimal because the runtime dependencies should be there only as available files, not installed products. Conan handles this situation better, thanks to referencing products in the cache. We should support building a package in an environment containing only its build dependencies while runtime dependencies are stored separately. The only way our meta-manager can combine packages is to install them, so we can provide a separate directory, e.g., `/mnt/dependencies/`, where rootfs of the runtime dependencies would only be present. That way, we can, for example, have a different version of a base library in the two contexts of build and dependencies.

In conclusion, each package will have a specific build context associated with it. Again, the repository is responsible for providing this information. The build context can be a URL for a direct download of an archive, an image tag, a Dockerfile, or meta-manager input.

2.4 Resolution

pacman uses a relatively simple algorithm for dependency resolution. When multiple choices are available, it asks the user for a resolution. This implies that pacman dependency resolution is incomplete.

Conan employs a similar strategy of incomplete resolution. It may try a few choices but also uses an essentially greedy algorithm that does not ensure resolution completeness.

2.4.1 Repositories

pacman utilizes the concept of repositories by allowing the user to specify a list of repositories and their mirrors in the `pacman.conf` configuration file. The archlinux ecosystem also provides a unique repository, the AUR, a single repository containing user-created packages. This repository cannot be configured directly as a pacman repository, as it does not distribute binaries but only package metadata and build scripts.

Conan also employs repositories with its proprietary protocol. It also accepts a configuration where multiple repositories can be put.

We can either 1) let each package manager combine their repositories or 2) let each package manager expose their repositories as a common interface. If we want resolution completeness, we cannot let the managers do the resolution. Therefore, we must choose option 2).

Each repository will provide package metadata in a common interface. We implement the so-called *repository drivers* to present the interface. A repository driver is a module that presents a specific manager's repository as a meta-manager repository. The meta-manager will then be configured with a list of repositories. Each repository will be associated with a driver in the configuration. We can still allow the user to pass driver-specific parameters.

2.4.2 SAT Solving

Let us consider a single repository for now. For a complete dependency resolution, we let its requirements be expressed as a propositional formula. Variables correspond to packages. A variable set to true means the package is present. Propositional formulas can be solved using SAT solvers, which are a decently developed area in computer science.

A typical SAT solver can return any satisfying model when the input formula is satisfiable. A pathological example could happen in a repository with only regular dependencies (p_1 implies p_2). The SAT solver could return an assignment of true to all variables. The user expects at least a (locally) minimal result, meaning there should be no "extra" package that could be removed and still produce a valid package set.

The name for a model minimal in the above-defined sense is a *prime implicant*. We will, therefore, need not only to solve the formula but to find one of its prime implicants. Fortunately, there are efficient algorithms capable of producing prime implicants.

Another issue with SAT solving is that defining an ordering on the models is difficult. As shown by the pathological example above, where the package set contains all packages, some models can be better than others. Prime implicants are not globally minimal in the sense of the set size. Moreover, even two same-sized package sets can be evidently of much different quality for the user. Take as an example a situation when a user wants to install `glibc` from the archlinux environment and has the AUR configured as well. Multiple packages provide the interface `glibc` in such cases. Among them are the regular `glibc` and a Git version `glibc-git`. The version from Git always builds since it is from the AUR. Furthermore, it is not the package the user intended to install in most cases.

One feature of modern SAT solvers is incremental solving. During this type of solving, the solver remembers learned information between invocations and uses it for a significant performance benefit. Between two invocations, the formula can either be expanded with additional clauses or so-called *assumptions* can be passed. Assumptions are partial assignments of variables that must hold for that invocation. Assumptions only make sense in incremental solving because if only one invocation was desired, they could be appended to the formula. We can use assumptions to try different preferential assignments before processing the pure formula.

To return to our example with `glibc`, we can use assumptions to block `glibc-git` from being assigned true before `glibc`. We do that by first trying to solve the formula with the assumption that `glibc-git` is false. If we allowed repositories to specify assumptions, for example, the AUR specifying the assumption that all of its packages are false, we could achieve a form of order on the models. A problem with this solution is that we would have to check all subsets of assumptions to get the best result in all cases. This would mean an exponential time complexity, which would not be good with thousands of packages. Still, for the typical use case, trying only one sequence of subsets, each formed from the previous by removing one package, will suffice. That way, we test each package in at least one configuration. We will discuss the implementation details of this approach in Chapter 4.

2.4.3 Repository Union

In the previous section, we assumed a single repository. However, the meta-manager must deal with multiple repositories combined by an operation unionizing the provided packages.

A problematic situation for the meta-manager is when multiple packages in different repositories provide one interface. When a package depends on such an interface, the formula must contain a subformula expressing that if the depending package is present, then any package implementing the interface can be present. A logical disjunction would express this. Since the formula for the depending package is only provided by the repository that contains it, the subformula cannot be stated directly because it does not have information about all providers from other repositories. One solution is to use a helper variable representing the interface. This works when the dependency on the interface does not specify any version requirement. But when a version requirement is present, we cannot encode it into the variable because the formula would have to contain clauses defining all version comparisons. However, this solution provides helpful insight because the repository has to state the dependency indirectly in any case.

The meta-manager repositories will provide their requirement formulas *untranslated*. Apart from the formula, the repositories will provide *translator data*. This data will be of a simple format: a mapping of symbols to groups. The naming is not coincidental, as translator data groups often correspond to package groups. The meta-manager will first combine translator data from all repositories using a simple dictionary analog to a set union. This combined data will then be used to translate the formula from each repository. Therefore, the information about interfaces is communicated between the repositories through the meta-manager using the translator data combination. Ultimately, the individual repository formulas can be combined using a simple conjunction.

The repository can state a *translator name* for each atom in the formula. This name will be mapped to a *translator* based on the user configuration of the meta-manager. Translators will be modules in the same way as repository drivers. They accept an atom (a requirement) from an untranslated formula and return a propositional formula with atoms being the package variables.

Translators allow repositories to specify requirements directly as stated by the metadata because the expression will be translated. Requirements can, therefore, be thought of as predicates on packages.

Here are two examples. When two packages provide the same virtual package, the translator data will contain the virtual package as a group and two symbols corresponding to the package providing the virtual package. When there are two versions in a package group, a translator group with those two packages as symbols is contained in the translator data.

2.4.4 Dependency Confusion Attacks

The vcpkg documentation website states the following:

> vcpkg determines the responsible registry (or overlay) before reaching out to the network. This prevents package Dependency confusion attacks because name resolution does not depend on any external state [11].

We can apply a similar approach. A difference in our application is that multiple identifiers have to be resolved. Note that by resolution here, we do not mean the dependency resolution but the process by which the meta-manager assigns an authority (e.g., repository) to an identifier (string).

To introduce name resolution in repository formulas, the formula has to be split into subformulas, each associated with a variable name. Fortunately, the repository formulas will always be conjunctions of implication defining package requirements. The left side of each implication is a package variable. Therefore, the repository can map simple strings to untranslated propositional formulas. Allowing the user to associate a regex with each repository will give each variable a predefined repository authority. The semantics are that a repository's implication is considered only when the left side matches the repository's regex *and* doesn't match any regex of the preceding repositories. Repository priorities are, therefore, still a feature, but they do not cause the dangerous dynamic name resolution.

Other identifiers that could cause the attack in our application are groups in translator data and interfaces in package metadata. Both of these can be solved analogously to package variables.

2.4.5 Updates

pacman uses the so-called sync databases. We regard this feature of pacman as very clever because the only time pacman communicates with the remote repository is during repository synchronization. This approach fits well into our design because we require the whole repository formula at once to perform complete dependency resolution. We will call this operation simply *update*.

Furthermore, some managers (Conan) are not designed with fetching all metadata at once in mind. This could make fetching the data in a naive approach very inefficient. Another solution would be to use caching. Caching solves the problem of inefficiency, albeit its transparency would still cause the first repository use to be significantly slower.

The second reason we need a separate step for updating the repository is that updates introduce *epochs* into our design. An epoch is a snapshot of the repository state, and it is a beneficial property to have controlled epochs because the meta-manager can assume that the repository state did not change when the epoch did not change.

Because of this, repositories need to expand their interface with two features. One is the operation of an update. The update will not and can not be directly accessed by the meta-manager. The repository's user might not be its maintainer and, therefore, can not initiate its update. However, this project's scope is broader, and a separate utility using the update interface has to be provided.

The second feature is the epochs. Each piece of data that can change between updates must be provided alongside an epoch corresponding to it. When the meta-manager notices an epoch change during its invocation, it knows an update has occurred, and data received prior are no longer valid.

3 Model

In this chapter, we construct a formal model and proof for some of the meta-manager's features. We will focus only on simplified dependency resolution. Dependency resolution is only a subset of the features required for a usable manager, but we regard it as the most complex and interesting part of the whole application. Other program components will not be formally modeled.

3.1 Mathematical Notation

Before constructing the formal model, let us first define all mathematical notation used in this thesis for better clarity.

- $\langle a_1, a_2, \ldots, a_n \rangle$ – a tuple containing elements $a_1, a_2, \ldots, a_n$
- $\mathcal{P}(X)$ – the power-set of set X
- $X \times Y$ – the Cartesian product of sets X and Y
- $[X \to Y]$ – the set of all functions from X to Y
- $[X \rightharpoonup Y]$ – the set of all partial functions from X to Y
- $\operatorname{dom}(f)$ – the set of all values x for which $f(x)$ is defined
- $\mathrm{VF}_{\mathbb{P}}$ – the set of all propositional formulas over the set of symbols $\mathbb{P}$

3.2 Common

First, we provide definitions we will use in both integrated managers' models. Conan and pacman deal with two types of elementary objects: package names and versions. In reality, they are all strings with a particular format, but that information is unnecessary for the model. We define the concept of a *manager universe*, which is the context in which a manager operates.

Definition 1 (manager universe)**.** A manager universe is a tuple $\langle N, V \rangle$ where:

- N is a set of package names
- V is a set of versions

Both managers allow packages to specify their requirements about the presence of other packages. Both support a format where the maintainer specifies a package name and an expression that constrains the permitted versions of the referenced package. This predicate on the versions can be expressed mathematically as a version subset.

Definition 2 (package requirement)**.** The set of all possible package requirements for a package name set N and a version set V is:

$$\rho(N, V) \coloneqq N \times \mathcal{P}(V)$$

3.3 pacman

A pacman repository only contains one version per package. Each package name is unique in a single repository and associated with exactly one version. Other information about the package is likewise known just from the package name. The repository can be, therefore, thought of as a function from a package name to information about the package. Not all possible package names are contained in a repository, so the function is only partial.

Important information for version resolution is the package's version, the virtual packages provided along with their versions, and the requirements: dependencies and conflicts.

The package name, version, and provided virtual packages form the interfaces of the package on which other packages can depend.

As pacman supports providing virtual packages without a version, we need to allow a special value to be set as the version of the provided package. Let us call that value *none*.

Definition 3 (pacman repository)**.** Let $V^+ := V \cup \{none\}$ for a version set V.

A pacman repository in universe $\langle N, V\rangle$ is a partial function

$$R : N \rightharpoonup V \times \mathcal{P}(N \times V^+) \times \mathcal{P}(\rho(N, V^+) \times \mathcal{P}(\rho(N, V^+))$$

For $R(n) = \langle v, p, d, c\rangle$:

- $\mathit{interfaces}_R(n) := \{\langle n, v\rangle\} \cup p$
- $\mathit{depends}_R(n) := d$
- $\mathit{conflicts}_R(n) := c$

Consider a hypothetical pacman repository with package `p` for which command `pacman -Si p` prints these lines:

```
Version         : v
Provides        : pr1 pr2=vpr
Depends On      : d1 d2=vd
Conflicts With  : c1 c2>=vc
```

Let V_c denote the subset of versions, for which comparison `>=vc` returns true. We can then model the example with the following:

$$R(\mathtt{p}) = \langle \mathtt{v}, \{\langle \mathtt{pr1}, none\rangle, \langle \mathtt{pr2}, \mathtt{vpr}\rangle\}, \{\langle \mathtt{d1}, V^+\rangle, \langle \mathtt{d2}, \{\mathtt{vd}\}\rangle\}, \{\langle \mathtt{c1}, V^+\rangle, \langle \mathtt{c2}, V_c\rangle\}\rangle$$

Note that requirements that specify a version expression are modeled with a set that does not contain the special value *none*. Only the requirements that do not specify any constraint on the version allow the interface to be provided with *none* version. This follows the pacman semantics.

Packages are used in sets. Only some sets of packages are correct in the sense of satisfying package requirements. To satisfy a package requirement, a package that provides that specified interface of a satisfactory version must be present in the package set.

Definition 4 (pacman requirement satisfaction)**.** Requirement $\langle r_n, r_v \rangle \in \rho(N, V^+)$ is satisfied by n in pacman repository R in universe $\langle N, V \rangle$ iff there exists a $v \in V^+$ such that $\langle r_n, v \rangle \in \mathit{interfaces}_R(n)$ and $v \in r_v$.

Requirement r is satisfied in package name set $P \subseteq \operatorname{dom}(R)$ from repository R iff there exists a package name $n \in P$ which satisfies requirement r in R.

Dependency is a package requirement that must be satisfied if the package is present. Conflicts can be thought of as inverted dependencies. When a package has a conflict requirement, the requirement must not be satisfied.

Definition 5 (pacman package set consistency)**.** $n \in P$ has requirements satisfied in $P \subseteq \operatorname{dom}(R)$ from pacman repository R iff all requirements in $\mathit{depends}_R(n)$ are satisfied in P and all requirements in $\mathit{conflicts}_R(n)$ are not satisfied in P.

A pacman package name set is consistent iff every package name in it has its requirements satisfied.

3.4 Conan

Conan differs from pacman in one important feature. A Conan repository can contain multiple versions of a single package. Conan requirements are simpler because there are no virtual packages and conflicts. No information is shared between two packages of the same name and different versions in the context of dependency resolution.

We can model a Conan repository similarly to a pacman repository. We will add one more mapping level, which will be package versions. The mapped-to value will only be a set of dependencies, as provided packages and conflicts are not part of Conan package metadata.

Conan does not use the special version value *none* as it is only used in virtual packages without a version specification.

Definition 6 (Conan repository)**.** A Conan repository in manager universe $\langle N, V \rangle$ is a partial function $N \rightharpoonup [V \rightharpoonup \mathcal{P}(\rho(N, V))]$.

Since Conan allows multiple versions of a package in a single repository, the concept of a package set becomes more complex. It still holds that there can be at most one version per package name in an actual package set. We can define a Conan package set as a partial function from package names to versions. For a package set to make sense, it needs only to map package names that are present in a repository to versions that are also present in that repository for that specific package name.

Definition 7 (Conan package set)**.** A Conan package set P from repository R is a partial function $N \rightharpoonup V$ where $\operatorname{dom}(P) \subseteq \operatorname{dom}(R)$ and $P(n) \in \operatorname{dom}(R(n))$ for every $n \in \operatorname{dom}(P)$.

Conan requirements directly specify the package name that should be present in the set. Their satisfaction is, therefore, simple to define.

Definition 8 (Conan requirement satisfaction)**.** $\langle r_n, r_v \rangle \in \rho(N, V)$ is satisfied in Conan package set P iff $P(r_n) \in r_v$.

Conan package set consistency is analogous to the pacman consistency. This means that all packages in the set have to have satisfied requirements.

Definition 9 (Conan package set consistency)**.** Package name n has requirements satisfied in package set P from Conan repository R iff all requirements in $R(n)(P(n))$ are satisfied in P.

A Conan package set is consistent iff every package name in it has its requirements satisfied.

3.5 Meta-manager

One of the meta-manager's goals is to provide complete dependency resolution. We achieve this using an SAT solver, meaning repositories must provide the dependency information as a propositional formula.

Another goal of the meta-manager is to combine repositories from multiple package managers. This is done using formula translation based on translator data.

To separate responsibilities, the exact semantics of the translation are given by modules separate from the meta-manager. Repositories provide a translator name with each value, and the meta-manager is configured to map translator names to specific translators.

Definition 10 (meta-manager universe)**.** A meta-manager universe is a tuple $\langle \mathbb{P}, \theta, \mathcal{R}, G, S \rangle$ where:

- $\mathbb{P}$ is a set of variables
- θ is a set of translator names
- $\mathcal{R}$ is a set of requirement values
- S is a set of symbols
- G is a set of groups

We can define translator data as a function from a group to a set of symbols associated with the group. It can be a complete function because the repository can define groups unrelated to it as mapped to an empty set.

To make the meta-manager less susceptible to dependency confusion attacks, the formula is provided as a set of subformulas, each denoted by a variable. The meta-manager can then filter based on the variable.

To allow repositories to use variables directly in the provided formula, we introduce a special symbol id, which denotes an identity translator.

Definition 11 (meta-manager repository)**.** A meta-manager repository in universe $\langle \mathbb{P}, \theta, \mathcal{R}, G, S \rangle$ is a tuple $\langle T, F \rangle$ where:

- T is a function $G \to \mathcal{P}(S)$
- F is a partial function $\mathbb{P} \rightharpoonup \mathrm{VF}_{(\theta \cup \{id\}) \times \mathcal{R}}$

The meta-manager needs to combine translator data from all repositories. The resulting data will contain the union of all groups, and each group will be mapped to the union of its contents from each datum.

Definition 12 (translator data union)**.** Let $T_1, T_2 \in [G \to \mathcal{P}(S)]$. $(T_1 \mathbin{\square} T_2)(g) \coloneqq T_1(g) \cup T_2(g)$.

We model translators as separate entities. They should accept requirement values and produce formulas over variables. However, the resulting formula is also dependent on translator data. We can, therefore, model them as functions from any translator data to a function that maps requirement values to formulas.

Definition 13 (translator)**.** A translator in universe $\langle \mathbb{P}, \theta, \mathcal{R}, G, S \rangle$ is a function $[G \to \mathcal{P}(S)] \to [\mathcal{R} \to \mathrm{VF}_{\mathbb{P}}]$.

To make specific managers accessible to the meta-manager, we must specify how to transform their repositories into meta-manager repositories.

For a pacman repository, the variables of the meta-manager universe have to include package names, as they identify specific packages. No helper variables are needed.

The only translator used by pacman is its own translator. Note that this does not have to be the case in all managers. Some managers might provide packages that depend on packages from different managers and could require different translators to specify those dependencies.

Values specified as requirements are the pacman requirements. pacman does not use the *id* translator, so we do not need package names as requirement values.

To make it possible for the translator to distinguish translator data groups, the driver prepends the special value pacman to the package name. If another manager provides a group with the same name, it can prepend its own special value so the groups do not mix.

In a pacman repository, the translator data groups are all possible interfaces the repository contains. That includes all names of packages in the repository and any virtual package provided by any package. That interface is mapped through the data to a set of symbols. The symbols are tuples of a package name and a version. The package name identifies the package that provides the interface. The version signifies the version at which the interface is provided.

The formula maps all variables corresponding to a package name in the repository. A package name is mapped to a conjunction of its requirements. Since the translator handles requirements, the repository can specify the requirements as they are. All dependencies must be true, while all conflicts must be false.

Definition 14 (pacman repository driver)**.** Let $\langle N, V \rangle$ denote some manager universe. Assume a package universe $U = \langle \mathbb{P}, \theta, \mathcal{R}, G, S \rangle$ where:

- $\mathbb{P} \supseteq N$
- $\theta \ni \textsf{pacman}$
- $\mathcal{R} \supseteq \rho(N, V^{+})$
- $G \supseteq \{\textsf{pacman}\} \times N$

- $S \supseteq N \times V^+$

The pacman repository driver is then the function *pacmandriver*, which maps a pacman repository to a meta-manager repository.

$$\mathit{pacmantranslatordata}(R)(\langle \textsf{pacman}, n\rangle) \ni \langle p, v\rangle \iff \langle n, v\rangle \in \mathit{interfaces}_R(p)$$

$$\mathit{pacmanformula}(R)(n) := \bigwedge_{d \in \mathit{depends}_R(n)} \langle \textsf{pacman}, d\rangle \wedge \bigwedge_{c \in \mathit{conflicts}_R(n)} \neg\langle \textsf{pacman}, c\rangle$$

$$\mathit{pacmandriver}(R) := \langle \mathit{pacmantranslatordata}(R), \mathit{pacmanformula}(R)\rangle$$

The pacman translator needs to handle translator data as defined above. A pacman requirement specifies an interface and allowed versions. This interface can be looked up in the translator data with the pacman prefix. The resulting values need to be filtered for satisfactory versions. The requirement can be satisfied by any of the package names of the filtered values. We can use a disjunction to achieve that semantics in a propositional formula. As the translator is a complete function, we must also handle cases where the input requirement is in a different format. In that case, we can return a false constant. That would make the resulting formula unsatisfiable when the unknown requirement is needed, which is the expected behavior.

Definition 15 (pacman translator)**.** For a manager universe $\langle N, V\rangle$, for every $r_n \in N$ and every $r_v \in V^+$:

$$\mathit{pacmantranslator}(T)(\langle r_n, r_v\rangle) := \bigvee_{\substack{\langle n,v\rangle \in T(\langle \textsf{pacman}, r_n\rangle) \\ v \in r_v}} n$$

For any other value, the translator returns $\bot$.

Next, we must define the transformation from a Conan repository to a meta-manager repository. The universe variables must contain tuples of package names and versions because, in Conan, we also need a version to identify a package uniquely.

Conan exclusively uses the Conan translator, and that must be reflected in the translator names.

Since Conan uses the *id* translator, the requirement values must contain the name-version tuples as well as all possible Conan requirements.

In Conan, interfaces are equal to package names. Therefore, groups are package names again prefixed with a special symbol. This means that pacman and Conan can provide a package with the same name.

The nonexistence of proper interfaces in Conan also causes the symbols to suffice with only a version. The translator data, therefore, associates each package name with all of its versions.

The formula maps all pairs of package names and versions contained in the Conan repository. For a specific package, the formula includes a conjunction of its dependencies and a subformula forbidding other versions of the same package to be present.

Definition 16 (Conan repository driver)**.** Let $\langle N, V \rangle$ denote some manager universe. Assume a meta-manager universe $\langle \mathbb{P}, \theta, \mathcal{R}, G, S \rangle$ where:

- $\mathbb{P} \supseteq N \times V$
- $\theta \ni \mathsf{Conan}$
- $\mathcal{R} \supseteq \rho(N, V) \cup N \times V$
- $G \supseteq \{\mathsf{Conan}\} \times N$
- $S \supseteq V$

A Conan repository driver is the function *conandriver* that maps a Conan repository to a meta-manager repository.

$$\mathit{conantranslatordata}(R)(\langle \mathsf{Conan}, n \rangle) \coloneqq \mathrm{dom}(R(n))$$

$$\mathit{conanformula}(R)(\langle n, v \rangle) \coloneqq \bigwedge_{d \in R(n)(v)} \langle \mathsf{Conan}, d \rangle \wedge \bigwedge_{\substack{v' \in V \\ v' \neq v}} \neg \langle id, \langle n, v' \rangle \rangle$$

$$\mathit{conandriver}(R) \coloneqq \langle \mathit{conantranslatordata}(R), \mathit{conanformula}(R) \rangle$$

The Conan translator is very similar to the pacman translator. Each Conan requirement is translated by a look-up in the translator data based on the required package name. Then, the versions are compared, and only the satisfactory package versions are selected for the disjunction. The main difference is that the variables in Conan are name-version pairs, so both must be included in the resulting formula. The translator also needs to return the false constant when it encounters an invalid format input.

Definition 17 (Conan translator)**.** For a manager universe $\langle N, V \rangle$, for every $n \in N$ and every $r_v \in V$:

$$\mathit{conantranslator}(T)(\langle n, r_v \rangle) \coloneqq \bigvee_{\substack{v \in T(\langle \mathsf{Conan}, n \rangle) \\ v \in r_v}} \langle n, v \rangle$$

For any other value, the translator returns $\perp$.

We will now define the semantics of formula translation in the meta-manager. Each translation should map a pair of translator names and requirement values to a formula. Since translators use translator data as a parameter, it is also a parameter in the meta-manager translation. The translator names must also be resolved so the translation accepts a mapping from translator names to translators. The mapping is a part of a meta-manager's invocation configuration.

The special translator *id* is a special case built into the meta-manager. It is simply the identity function on any requirement value. The translator name mapping and translator data are not considered with this translator.

An atomic formula is translated by invoking the correct translator on the requirement value specified by the translator name and the name mapping.

Compound formulas are translated recursively using the same operator in the result as in the input.

Definition 18 (meta-manager formula translation)**.** Let τ denote a mapping from translator names to translators.

For every requirement value $r \in \mathcal{R}$:

$$\mathit{translate}(\tau, T, \langle id, r\rangle) := r$$

For every atomic formula $\langle t, r\rangle \in \theta \times \mathcal{R}$:

$$\mathit{translate}(\tau, T, \langle t, r\rangle) := \tau(t)(T)(r)$$

For any two subformulas φ and ψ and an operator $\circ$:

$$\mathit{translate}(\tau, T, \varphi \circ \psi) := \mathit{translate}(\tau, T, \varphi) \circ \mathit{translate}(\tau, T, \psi)$$

The last part of the model is the logic by which the meta-manager creates the final formula that can be used as an input for a SAT solver. In short, the meta-manager combines formulas from all repositories and translates them. The repositories are part of the meta-manager configuration.

First, the meta-manager creates the final translator data using the special union-like operator on data from all repositories. This whole data must be provided to each translation as the meta-manager contains no translation logic.

The repositories provide the formula as a mapping from variables. This allows the meta-manager to select only a subset of the formula to be used. It is also necessary for each variable to be governed by only one repository. Therefore, the meta-manager needs to be configured with a mapping from variables to repositories.

Formula combination is done trivially by conjunction. The whole variable-to-repository mapping is iterated, and the subformulas corresponding to the repository and variable are combined. The meta-manager interprets each mapping of a variable to a formula by a repository as an implication from the variable to the formula.

Definition 19 (meta-manager formula)**.** Assume a universe $\langle \mathbb{P}, \theta, \mathcal{R}, G, S\rangle$. Let M be a mapping from variables to meta-manager repositories, $[R]$ a set of meta-manager repositories, and τ a mapping from translator names to translators.

A meta-manager formula is then the function:

$$\Phi(M, [R], \tau) := \bigwedge_{\langle T,F\rangle \in [R]} \ \bigwedge_{\substack{\langle v,\varphi\rangle \in F \\ M(v) = \langle T,F\rangle}} v \rightarrow \mathit{translate}(\tau, \bigsqcup_{\langle T,F\rangle \in [R]} T, \varphi)$$

Theorem 1 (meta-manager is correct)**.** *Assume a manager universe $U_M = \langle N, V\rangle$ and a meta-manager universe*

$$\langle N \cup N \times V, \{\mathsf{pacman}, \mathsf{Conan}\}, \rho(N, V^+) \cup N \times V, N \times V^+ \cup V, \{\mathsf{pacman}, \mathsf{Conan}\} \times N\rangle$$

Let $[R]_P$ be the set of all pacman *repositories over U_M and $[R]_C$ the set of all* Conan *repositories over U_M. Let*

$$[R] \subseteq \mathcal{P}(\mathit{pacmandriver}([R]_P) \cup \mathit{conandriver}([R]_C))$$

$$\tau(\textsf{pacman}) \coloneqq \mathit{pacmantranslator} \wedge \tau(\textsf{Conan}) \coloneqq \mathit{conantranslator}$$

Let $\mathit{trues}(m)$ denote the set of all variables set to true in model m, $R_P(n) \coloneqq M(n)(n)$, $R_C(n)(v) \coloneqq M(\langle n, v\rangle)(n)(v)$.

Then for every mapping M and model $m \models$, it holds that $\mathit{trues}(m) \cap N$ is consistent in R_P and $\mathit{trues}(m) \cap N \times V$ is consistent in R_C.

Proof. Let M be any variable to repository mapping and $\varphi \coloneqq \Phi(M, [R], \tau)$. As the repositories in $[R]$ come only from pacman and Conan repository drivers, we have

$$T \coloneqq \bigsqcup_{\langle T', F\rangle \in [R]} T'$$

$$T(\langle \textsf{pacman}, n\rangle) = \mathit{pacmantranslatordata}(R_P)(\langle \textsf{pacman}, n\rangle)$$

$$T(\langle \textsf{Conan}, n\rangle) = \mathit{conantranslatordata}(R_C)(\langle \textsf{Conan}, n\rangle)$$

First, we will prove pacman consistency. Let $P_P \coloneqq \mathit{trues}(m) \cap N$. P_P being consistent in R_P means that all package names in P_P have their requirements met. Let p denote any such package name, $\langle d, v_d\rangle$ any of its dependencies, and $\langle c, v_c\rangle$ any of its conflicts. Then

$$\varphi \models p \to \mathit{translate}(\tau, T, \langle \textsf{pacman}, \langle d, v_d\rangle\rangle)$$

$$\varphi \models p \to \neg\mathit{translate}(\tau, T, \langle \textsf{pacman}, \langle c, v_c\rangle\rangle)$$

From the definitions of *pacmantranslatordata* and *pacmantranslator* we then have:

$$\varphi \models p \to \bigvee_{\substack{\langle d, v\rangle \in \mathit{interfaces}_{R_P}(n) \\ v \in v_d}} n$$

$$\varphi \models p \to \neg\left(\bigvee_{\substack{\langle c, v\rangle \in \mathit{interfaces}_{R_P}(n) \\ v \in v_c}} n\right)$$

Since $p \in \mathit{trues}(m)$, the right-hand sides of the implications are true in m. Therefore, there exists a package name $n_c \in \mathit{trues}(m)$ which satisfies the dependency $\langle d, v_d\rangle$ and similarly there is no package in $\mathit{trues}(m)$ which satisfies $\langle c, v_c\rangle$. Since $P_P \coloneqq \mathit{trues}(m) \cap N$, dependencies of p are satisfied in P_P, and none of its conflicts are satisfied in P_P. This means that any package name in P_P has its requirements satisfied and P_P is therefore consistent in R_P.

Let $P_C \coloneqq \mathit{trues}(m) \cap N \times V$. To prove Conan consistency, we first need to prove that P_C is a partial function from N to V. That requires that for every package name $n \in N$, there are no two distinct versions $v, w \in V$ for which $\langle n, v\rangle \in \mathit{trues}(m)$ and $\langle n, w\rangle \in \mathit{trues}(m)$. Consider any $n \in N$ for which $\langle n, v\rangle$

belongs to *trues*(m). After translating the meta-manager formula using the *id* translation, we have:

$$\varphi \models \langle n, v\rangle \to \bigwedge_{\substack{v' \in V \\ v' \neq v}} \neg\langle n, v'\rangle$$

Since $\langle n, v\rangle$ is set to true in model m, no other $\langle n, w\rangle$ can also be set to true because that would make the implication false. P_C is, therefore, a partial function, and it is valid to consider whether it is consistent as a Conan package set.

We can now prove the Conan consistency. P_C being consistent in R_C means that all package names in P_C have their requirements met. Consider any package name n in P_C. Let $v \coloneqq P_C(n)$ and $\langle d, v_d\rangle$ denote any dependency in $R_C(n)(v)$. We then have

$$\varphi \models \langle n, v\rangle \to \mathit{translate}(\tau, T, \langle \mathsf{Conan}, \langle d, v_d\rangle\rangle)$$

From the definitions of *conantranslatordata* and *conantranslator* we translate the formula as follows:

$$\varphi \models \langle n, v\rangle \to \bigvee_{\substack{v' \in \mathrm{dom}(R(d)) \\ v' \in v_d}} \langle d, v'\rangle$$

Since $\langle n, v\rangle$ belongs to *trues*(m), the right-hand side of the implication is true in m. That means that $\langle d, v'\rangle$ is set to true in m for some version v', which satisfies the version requirement v_d. Since $P_C \coloneqq \mathit{trues}(m) \cap N \times V$, all dependencies of n are satisfied in P_C which makes P_C consistent in R_C. □

4 Implementation

In this chapter, we will first discuss the implementation of the modules containing the integrated managers' semantics. Then, we will review the most interesting parts of the meta-manager implementation.

4.1 pacman

We will use the libalpm pacman backend through the `pyalpm` Python bindings library. To separate pacman packages from Conan packages with the same name, we will prefix each identifier (group, interface, package variable) with `pacman-`.

4.1.1 Repository Driver

Translator Data

The driver will return several group-symbol pairs for each package. The following applies to each package.

Each package provides its name as one of its interfaces. One pair's group will be the package's name. To support version comparisons, the symbol must contain the version. The package name is already included in the group, so that is all the information the symbol needs to contain.

Another type of package interface is a provided virtual package. For each virtual package, the driver will return one pair. The pair's group will always be the virtual package name, as that is what is referred to in the requirements. If the provided package has a version specification, the symbol will contain that version. The symbol will also include the package providing the interface. Note that the group contains the interface name, not the provider's name.

Formula

For all requirements returned in pacman formulas, the translator is the `pacman` translator. The package variables are formed as the package group name followed by its pacman version and release.

The two types of requirements in pacman are dependencies and conflicts. To recapitulate dependency semantics, if the depending package is present, the dependency must be satisfied. Note that because of formula translation, we can express the semantics just as we stated. We do not yet have to specify the formula in terms of package variables. Therefore, for each dependency, we return an atomic requirement formula containing the dependency as it is—in its string form. We decided to conservatively wait with parsing until the latest possible instance.

Conflicts can be handled analogously. The semantics are similar but with an added negation. However, there are small details we need to pay attention to. Sometimes, two packages provide the same interface in a convoluted way. It is possible for a package to provide a virtual package with the same name as another package. In that case, it is probable that the two packages are each other's replacements and that a conflict will also be specified. The conflict, therefore, needs to be specified with the interface, i.e., one of the package names. This means we must handle cases where a package conflicts with its name. We solved this issue by providing the package name in the requirement itself. Conflicts, therefore, specify that the requirement must *not* be satisfied, *except* when the satisfying package is the conflicting package itself. This exception avoids a trivially unsatisfiable formula. The conflict requirement formula will be a negation of a compound atom containing the conflict specification and the conflicting package.

Package Detail

Package detail is supposed to return interfaces and interface dependencies of a package. It is straightforward to define these, as pacman already deals with these terms. The only modification that needs to be done is the removal of versions. The versions serve to determine valid package sets, but when dealing with interfaces, the package set is already fixed.

Product Interface Information

Since pacman repositories distribute binary packages, the package version uniquely identifies the ABI. It is also the only information pacman provides to the consumer. The PII will, therefore, contain only the version.

Build Context

pacman fetches packages by downloading them. The only interface libalpm provides for downloading packages is setting the repository mirrors and downloading the archive to the cache. If the driver were remote, we would have to download the package to the server and provide it from there.

Fortunately, archlinux provides package archives behind a URL-based HTTP interface. We can search for archives there by package name and version. The URL created from the specific package will be its build context.

Update

Repository update implementation is very straightforward because libalpm directly provides an interface for synchronizing the sync databases.

4.1.2 Translator

Assumptions

The goal is to prioritize regular pacman packages over those from the AUR. We must disambiguate the AUR packages in the translator data. This is the responsibility of the AUR's repository driver.

The assumptions need to be all AUR package variables set to false. The performance of the resolution depends on the number of assumptions. To make their number smaller, we will leverage the translator data and only return an assumption for those packages that provide an interface also provided by a regular pacman package. If a package provides an interface unique to the AUR, there is no point in assuming that the package is not present, as the only way the variable would be set to true in the prime implicant is that it was, in fact, necessary.

Translation

A requirement specifies a name and an optional version specification. Thanks to the structure of the translator data we created, the name can be looked up as a group. The symbols in the group specify all the possible packages that can satisfy the requirement's name. This is true because of the way we structured the translator data: each symbol can be mapped to a single package. Whether they can satisfy the version must be determined. Since we did not parse the requirement in the repository driver, we need to include the parsing logic here. After parsing, we can pass the versions from the requirement and the symbol to the libalpm's interface for comparing versions. Together with the parsed type of comparison from the requirement, we can filter the packages that satisfy the requirement. Since any package is valid, we return a disjunction of their variables.

If an exclusion requirement is received, we must exclude the stated package from the disjunction.

4.1.3 Installer

pacman invokes arbitrary commands in the installation directory. When the user passes the option `--root`, it uses `chroot` to invoke the command as if in the context of the directory. `chroot` has limitations and is not at all secure. This is why we decided to hijack the libalpm hook invocation.

We must execute the commands in a container with its rootfs set to the installation directory. Therefore, we want to disable libalpm's regular hook invocation. At the same time, we need to determine somehow which commands libalpm would run.

We decided to do this with the `LD_PRELOAD` approach. We overwrite the `chroot` and `execv` system calls. The `chroot` is overridden to a noop. `execv` needs to read the command and the arguments and pass it to the meta-manager. To use `LD_PRELOAD`, we must invoke a separate executable from the manager. The executable needs to communicate with the main application; we use UDS for this. The UDS server in the meta-manager listens for commands given by the executable and executes them in the container.

4.2 AUR

The AUR is a repository of pacman compatible packages. It uses pacman semantics for requirements. Built packages are installed with pacman as well. As a result, the only module that needs to be implemented to integrate AUR into the meta-manager is its repository driver.

AUR identifiers will share a common prefix with pacman, `pacman-`, as they semantically form the same package namespace.

4.2.1 Repository Driver

The AUR has an excellent interface for downloading all package metadata at once. The format of this data is a single JSON file. It is a suitable format for human readability and parsing, but it would be too slow to parse all the data each time the repository driver needs to access it. Because of that, we decided to store the metadata in a file-based SQL database. Since the data is relational, a relational database is a good fit.

Translator Data

The translator data provided by the AUR driver are essentially the same format as the pacman data. The main difference between these two drivers lies in the interface through which the package metadata is accessed. However, one slight difference is mentioned in the pacman translator implementation. To enable assumptions, the AUR has to mark its symbols with a flag.

Formula

The algorithm for creating the formula follows the same logic as the pacman one.

Package Detail

The interfaces are also of the same semantics and are hence analogous to the pacman implementation.

Product Interface Information

Communicating the ABI of packages built with AUR is a complex problem. Unfortunately, a PKGBUILD can download a different source at different times. The ultimate solution would be to be able to determine the source code version. One idea is to use the hash of all source files downloaded during the build process. Since the current Arch build system does not provide the feature anyway, we decided on the limited implementation without consideration of the source code version.

As a result, the PII will contain the version as stated by the package metadata and the versions of all runtime dependencies. As AUR packages only depend on other AUR packages and regular pacman packages, and both share a common PII format, we can determine the versions of dependencies quickly.

Beware that this simplification currently causes a situation where a new source code version without a bump to the package metadata version will not cause a change to the PII and the old product is considered equivalent. The only fix would be to remove the old product from the cache. This is also the current instructions given to the users of the AUR, so it would not be a downgrade in function to use the meta-manager.

Build Context

The main difference between AUR and pacman packages is that AUR packages are built from source. The package will be built in a container whose image was created by the meta-manager. The packages that need to be present are the regular and build dependencies. The build context image's requirements will be the conjunction of all runtime and build dependencies of the built package.

Package base-devel is an implicit build dependency for all packages. AUR provides a separate Git repository for all its packages that contain their source, so we will also require the git package.

The script to build the package in the container must first clone the Git repository. makepkg does not work when run as the root user, so we need to choose a different user. User nobody is a good choice, as the build should not require special privileges. Furthermore, nobody is always present in a system. We discovered that the container rootfs can not be deleted if nobody owns some files. At the end of the build script, we need to chown the files back to the root user.

Update

During repository update, the JSON with package metadata must be downloaded, parsed, and stored in the database.

A performance benefit could be derived if an iterative JSON parser was used, but we opted not to optimize prematurely as no performance issues were detected in this phase. On the contrary, considering the number of packages in the AUR, the performance of this step is good (seconds). This is probably thanks to the metadata bundled together in a single file.

4.3 Conan

Conan is a separate package manager from pacman, so we need to implement each module type for its integration.

The manager is a Python application, so we can use it as a library. It also provides a proper API, which we will prefer when possible. The API is also currently not stable, so there is nonetheless the issue of losing compatibility with a future Conan release.

Conan identifiers will use prefix conan-.

4.3.1 Repository Driver

Translator Data

Interfaces in Conan are just package group names. So, for each package, we will return one symbol mapping. The group will be the package group name and the symbol will be the Conan package version and revision, which combined form the package version. To support complete dependency resolution, the goal is to express all possible versions from a package group in the translator data.

Formula

The package variables must uniquely identify a package. So, they will contain the package group name with its Conan version and revision.

Conan utilizes only ”positive” requirements, so for each runtime requirement, we will return a requirement formula stating that requirement. Since we use Conan as a Python library, the requirement is partially parsed, and we will pass it in the formula atom in that form. The translator for the regular requirements is conan.

Conan also supports system requirements. Since pacman is supported as one ofConan's system managers and we also support it, we can translate each system requirement as an atomic requirement formula with the translator set to `pacman`.

We also found that there is no need to explicitly ban multiple versions of the same package, as that is resolved by computing prime implicants.

Package Detail

As already stated, interfaces in Conan are just package group names. Each package, therefore, provides an interface with its package group name. It also depends on interfaces according to its runtime Conan requirements. Each system requirement can also be parsed as a requirement on a pacman interface. We have to remove the optional version specification.

Product Interface Information

Calculating PII for Conan packages is straightforward because the Conan concept of package ID maps directly to our concept of PII. In addition to package ID, we must also provide the Conan version and revision. This is because the ABI depends on the exact dependency versions provided when building the package.

We could also optionally provide the system requirements versions, but the original Conan implementation does not do this, and we decided on a simpler implementation.

Build Context

Conan packages must be built similarly to AUR packages. The build context will be a container run on a meta-manager-created image. The requirements for this image are analogous to the AUR ones. The build requires all build requirements and, of course, Conan itself, as the manager handles the build. Fortunately, it is provided as a package in AUR under the name `conan`, so we can also specify this requirement. In addition to Conan, we determined more than ten other unstated requirements for building Conan packages.

To build, `conan install` has to be executed with the option to enable the build of the required package. After the build completes successfully, we export the archive containing the product from the cache using `conan save`.

Update

Conan automatically downloads package metadata into the cache when handling any package. We must explicitly fill the cache with all available packages during the update.

Unfortunately, the interface is designed in a way that allows us to only search for package groups. A separate remote call has to be made for each package group to determine all of its versions. This could still be handled well if the remote calls were asynchronous, but they are not. This slows down this procedure and is an unfortunate design decision of Conan developers.

4.3.2 Translator

Assumptions

Conan does not require any package prioritizing.

Translation

We employ the same logic as with the pacman translator. A requirement names a group we will look up. The symbols in this group determine all packages from this group. The symbols, and therefore the packages, need to be filtered using the version specification. Conan unfortunately does not provide the version comparison logic in its API, so we had to use an undocumented function found through a manual search in the Conan source code.

4.3.3 Installer

As was already stated, installation of Conan packages comes down to just putting the product into the cache. As Conan build produces the exported product archive, we must import it into the cache.

4.4 Meta-manager

4.4.1 Technology Selection

The meta-manager will be written in Python. The module interfaces will be implemented as Python interfaces because it is the simplest option. Better options will be discussed in Future Work.

Containerization will be targeting both Docker and Podman where possible, but we will prefer Podman if necessary due to its developer friendliness.

All structured data will have the JSON format. This applies to configuration files and standard inputs. We chose JSON because of its popularity, ability to express arbitrary structured data, and ease of use from Python.

In places where the project requires a database, we will choose the most straightforward option to save development time. We will use the `sqlitedict` Python library when sufficient. We will use an SQL database only when necessary, with our choice being the SQLite database for its ease of use. An advantage of both of these solutions is that they are file-based.

4.4.2 Containerization

If the meta-manager were invoked inside a container, it would be preferable not to create containers in containers, as this is a problematic aspect of containerization. Since the meta-manager must be able to run containers with custom rootfs, the containers cannot be started before the meta-manager is invoked. The meta-manager itself must control the invocation of the containers. One method to avoid running containers in containers is communicating with a containerizer through a socket mounted inside the container.

A problem with this approach is that the rootfs and mounts location paths must be passed relative to the containerizer. A containerized meta-manager cannot arbitrarily access the host filesystem. Therefore, a shared directory with a known location to both the containerizer and the containerized meta-manager must be mounted in the meta-manager container. The meta-manager must be configured with the path to that directory from its and the containerizer's point of view to pass the paths correctly.

Giving the meta-manager direct access to the containerizer might be a security risk. Another option would be to create a server application designed specifically for running containers needed in the meta-manager.

4.4.3 Prime Implicants

Many algorithms for calculating prime implicants were developed [12] [13] [14]. However, when searching for an actual implementation of the algorithm, the only SAT solver providing it we found was the Sat4j. The problem with Sat4j is that it is implemented as a Java library. One consequence is that we have to invoke Java from Python. It is not as simple as we would like. Since we use containerization anyway, we decided to invoke Sat4j as a container. We created the image with the Sat4j code and communicate with the container using file-based IO, where the files are located in predefined locations.

Another issue with prime implicants is that they are a byproduct of a standard solver invocation. This means that the whole solver needs to run in the Java container. Performance is, therefore, degraded, but we found it is not a problem with typical input sizes.

4.4.4 Assumptions

We have already determined that it is sufficient to try assumptions in a single sequence of subsets where we remove one assumption in each step. Experimentally, we concluded that for a typical configuration with archlinux official repositories, the AUR and conancenter enabled, we can invoke the solver approximately a hundred times per second. The number of assumptions the AUR repository contains is around a hundred thousand. Therefore, it is unfeasible to implement the algorithm as stated directly.

We will leverage an essential fact about testing assumptions. When testing multiple assumptions, getting a `SAT` result means that *all* of the assumptions are valid. Getting `UNSAT` means the negation: *some* of the assumptions are *not* valid. We already use this fact to end the algorithm: we stop when we receive `SAT`. We can employ a divide-and-conquer approach if we apply this fact to the whole search. We optimistically hope that large chunks of assumptions are valid. Getting a `SAT` confirms this belief, and the algorithm ends. If we receive `UNSAT`, we divide the problem in half and recursively search in both halves.

The worst-case input for this algorithm is a list of assumptions in which every other assumption is invalid and must be left out from the final set. It is the worst case because the algorithm has to descend all the way in each branch. The time complexity is linear to the number of assumptions, which is the same complexity the naive implementation would have. However, most assumptions would be valid in a typical input because the user's relatively small requirements would not influence whether most packages must or must not be present. We did not derive any more asymptotic time complexities, but our divide-and-conquer implementation handles typical inputs in a few seconds instead of the expected tens of minutes.

4.4.5 Repository Data Caching

The meta-manager needs to leverage caching to gain a performance boost. This only affects repositories, as they are the only part of the application designed to be communicated with remotely through the network. As a byproduct, caching allows the repository driver to employ a direct and inefficient implementation, as they can rely on client caching.

Conclusion

Let us first review the goals set in the Introduction.

Multiple managers are integrated into the meta-manager. These are namely pacman and Conan with the officially unsupported AUR integrated as well. The repository union is achieved mainly through a mechanism called formula translation.

The meta-manager architecture is modular. There are multiple points in the logic of the meta-manager where the exact behavior is realized by invoking a separate module with a clearly defined interface. The modules are shown in blue in Figure 2.1. Project expansion for an additional package manager can be achieved by implementing the correct modules and having the users use them.

The basis of the application's design is that the result of the meta-manager invocation is a rootfs directory. We even implemented a better behavior for pacman hooks by running the commands in a container instead of a `chroot`.

We achieve complete dependency resolution by employing SAT solving. Utilizing prime implicants is necessary for practical results. We also implemented our own simple divide-and-conquer algorithm for preferential package selection using incremental solving and assumptions.

The meta-manager uses a mechanism preventing dependency confusion attacks heavily inspired by vcpkg. In this way, we tried to solve some security concerns in package management.

Our meta-manager can build from the source and provides multiple mechanisms to support building. The package builds are realized in containers created by the meta-manager itself. The product ABI can be communicated to consumers using a concept inspired by the solution in Conan. Binary caching uses the same mechanism to determine product equivalence.

The application can also be run in a container. This has two benefits. One is that the usage of the application might be easier for some users since containers should work "out of the box". The second benefit is that the project is, therefore, prepared for itself being packaged and used through itself. We explored this concept in the Section 1.4.2.

We also developed a mathematical model of dependency resolution as implemented by the meta-manager to provide formal ground for the project. It helped design the meta-manager more rigorously as it uncovered multiple design weak spots. A particular example is the implications of inverting the relationship between identifiers and repositories to prevent dependency confusion attacks.

Overall, the project fulfilled the set goals. However, we still consider it a proof-of-concept application. Package managers provide many more features we did not cover, and the application has multiple usability issues. Despite this, we believe the project sets a valuable foundation for further exploration of non-traditional approaches to package management.

Future Work

This section explores two areas of the meta-manager that could be improved upon in its current form. However, many more possibilities could be explored after these two.

SAT Solving Improvements

The user typically wants to invoke the meta-manager multiple times with the same formula. This is because the formula only changes with repository updates. To improve this, we could exploit this fact and preprocess the formula. Several different preprocessing techniques have been developed. The ones we can use must be able to return models in terms of the original formula.

CDCL solvers learn clauses during their invocation. These learned clauses are used in incremental solving to improve performance. When the meta-manager is invoked with the same formula multiple times, the subsequent invocations could use the learned clauses. However, we did not find any way to extract the learned clauses from `Sat4j` nor any other solver.

A Custom Package Manager

To fully utilize the meta-manager's features, it would also be beneficial to implement a custom-built package manager for the project. Because of its modularity, this addition would not be a single application but a collection of modules.

For example, a custom repository driver could support simple `Git` or filesystem-based repositories. The `pacman` translator could probably be used even with the custom manager. A custom installer could be developed to avoid the complexity of `pacman` packages and installation hooks. It could simply unpack archives into the destination directory.

Bibliography

1. Spinellis, Diomidis. Package Management Systems. *IEEE Software*. 2012, vol. 29, no. 2, pp. 84–86. Available from doi: `10.1109/MS.2012.38`.
2. Foundation, Standard C++. *2022 Annual C++ Developer Survey "Lite"* [online]. 2022. [visited on 2024-07-12]. Available from: `https://isocpp.org/blog/2022/06/results-summary-2022-annual-cpp-developer-survey-lite`.
3. Foundation, Standard C++. *2023 Annual C++ Developer Survey "Lite"* [online]. 2023. [visited on 2024-07-12]. Available from: `https://isocpp.org/blog/2023/04/results-summary-2023-annual-cpp-developer-survey-lite`.
4. Foundation, Standard C++. *2024 Annual C++ Developer Survey "Lite"* [online]. 2024. [visited on 2024-07-12]. Available from: `https://isocpp.org/blog/2024/04/results-summary-2024-annual-cpp-developer-survey-lite`.
5. Miranda, André; Pimentel, João. On the use of package managers by the C++ open-source community. In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. Pau, France: Association for Computing Machinery, 2018, pp. 1483–1491. SAC '18. isbn 9781450351911. Available from doi: `10.1145/3167132.3167290`.
6. Hauser, Henrik. *Hardening the Software Supply Chain: Developing a System to Prevent Dependency Confusion Attacks in Cloud Based Continuous Integration and Deployment Processes*. 2022. MA thesis. Hochschule Wismar.
7. Team, Arch Linux. *ArchWiki* [online]. 2024. [visited on 2024-07-12]. Available from: `https://wiki.archlinux.org/`.
8. JFrog. *Conan 2 - C and C++ Package Manager Documentation* [online]. 2024. [visited on 2024-07-12]. Available from: `https://docs.conan.io/2/`.
9. Taeumel, Marcel; Lincke, Jens; Rein, Patrick; Hirschfeld, Robert. A pattern language of an exploratory programming workspace. In: *Design thinking research: Achieving real innovation*. Springer, 2022, pp. 111–145.
10. Cox, Russ. *Minimal version selection* [online]. 2018. [visited on 2024-07-12]. Available from: `https://research.swtch.com/vgo-mvs`.
11. Microsoft. *vcpkg documentation* [online]. [visited on 2024-07-12]. Available from: `https://learn.microsoft.com/en-us/vcpkg`.
12. Marques-Silva, Joao; Janota, Mikoláš; Mencia, Carlos. Minimal sets on propositional formulae. Problems and reductions. *Artificial Intelligence*. 2017, vol. 252, pp. 22–50.
13. Palopoli, Luigi; Pirri, Fiora; Pizzuti, Clara. Algorithms for selective enumeration of prime implicants. *Artificial Intelligence*. 1999, vol. 111, no. 1-2, pp. 41–72.
14. Déharbe, David; Fontaine, Pascal; Le Berre, Daniel; Mazure, Bertrand. Computing prime implicants. In: *2013 Formal Methods in Computer-Aided Design*. IEEE, 2013, pp. 46–52.

A Attachment

The thesis contains one ZIP attachment. It includes all software developed as part of the thesis.

The project is also hosted at `https://github.com/papundekel/PPpackage`.

A.1 Structure

The structure of the attachment is as follows:

- `src/` – source files for the meta-manager, modules, and helper libraries
- `examples/` – files to help try out the project; not part of the application itself
  - `input/` – inputs for the meta-manager
  - `metamanager/` – configurations of the meta-manager; contains configurations for both native and containerized invocation
  - `update/` – configurations for repository updates; configured for compatibility with examples in `metamanager/`
  - `project/compressor/` – project from the Conan tutorial modified for our project
- `Dockerfile` – Dockerfile defining all images needed for containerized invocation
- `image-build.sh` – script for building the images more easily

A.2 Usage

The application can be used either natively or containerized. We recommend the containerized approach, as it is simpler. Therefore, only the containerized invocation is documented and part of the attachment. All scripts are designed to be run from the top-level directory.

The invocation consists of two steps. First, repositories need to be updated (created on first use). Then, the actual meta-manager configured to use these repositories can run.

If a command produces any unexpected behavior or errors, please first consult Section A.2.6 at the end of the appendix.

A.2.1 TLDR

```
$ podman system service --time=0 &
$ ./image-build.sh podman solver
$ ./image-build.sh podman metamanager
$ ./image-build.sh podman updater
$ ./examples/update/containerized/update.sh podman
$ ./examples/metamanager/containerized/run.sh podman <$input
```

The variable `input` could be set to, for example, `examples/input/iana-etc.json`.

A.2.2 Containerizer

The project is heavily container-oriented and, therefore, requires a containerizer. Although the project supports Docker in most use cases and our GitHub Actions CI uses Docker successfully, we do not deem the support sufficient. Therefore, Podman is required. Only versions 4 and 5 were thoroughly tested.

The attached examples use the Compose Specification V2 for easier container configuration. Therefore, you need the `docker-compose` command version 2 or `podman-compose`. `podman-compose` can be installed via pip.

Note that `podman compose` and `podman-compose` are not the same. `podman compose` is just a driver calling either `docker-compose` or `podman-compose`. `docker-compose` has priority if installed. If both composers are installed, the `PODMAN_COMPOSE_PROVIDER` variable can be used to set which one to use.

The meta-manager also runs containers, so it needs access to the Podman daemon socket. The daemon can be run either through Podman or as a systemd service.

```
$ podman system service --time=0 &
$ # or
$ systemctl --user enable --now podman
```

A.2.3 Container Images

For containerized invocation, we need to obtain the images the application requires. To build them on your machine, use the following:

```
$ ./image-build.sh podman solver
$ ./image-build.sh podman metamanager
$ ./image-build.sh podman updater
```

It is normal for these commands to take a long time.

A.2.4 Update

```
$ ./examples/update/containerized/update.sh podman
```

This script initializes or updates package repository databases in your home directory. `$HOME/.PPpackage` is the directory containing all application data. This command needs to run only once in a while, i.e., when new package versions are desired.

It takes a few minutes for this command to finish, as the Conan implementation of package metadata fetching is not optimized for our use case.

A.2.5 Meta-manager

```
$ ./examples/metamanager/containerized/run.sh podman <$input
```

This script will invoke the meta-manager. The script is set up so the image rootfs are created at `./tmp/root`, and the generators are put inside `./tmp/output/generators`. You can run this command any number of times without performing an update after the repositories are initialized at the beginning.

The `input` variable needs to be replaced with a path to an input. The following is the complete list of prepared example inputs with paths relative to the `examples/` directory. They are ordered with rising complexity as we perceive it.

- `input/iana-etc.json`
- `input/glibc.json`
- `input/sh.json`
- `project/compressor/requirements.json`
- `input/conan-build.json`
- `input/conan-conflict/PP.json`

Beware that the first invocation of the meta-manager after the repositories are updated (the previous step) needs to cache repository formulas. Therefore, it also takes a few minutes to complete the dependency resolution.

Note that the meta-manager builds the requested packages, and the builds are done in container images recursively created by the meta-manager. This means that more complex inputs take significantly longer. However, the meta-manager uses caching, so no work is done twice.

`conan-conflict`

Conan does not offer complete dependency resolution, while our project does. This difference can be seen in the example `conan-conflict`. The meta-manager can resolve the dependencies while Conan cannot (at least until version 2.5).

Conan should output the following:

```
ERROR: Version conflict: Conflict between libpng/1.6.42
and libpng/1.6.43 in the graph.
Conflict originates from qt/6.4.2
```

`project/compressor/`

Conan tutorial contains a simple example with a library consuming the zlib package. We copied and modified this example to showcase how the meta-manager usage compares to current practices.

First, it is necessary to create the image rootfs. This image will be used as a build container for the example project.

```
$ ./examples/metamanager/$method/run.sh \
    <examples/project/compressor/requirements.json
```

The output is again created in the `./tmp/` directory. To build the project, we prepared a script. It needs the paths of the image rootfs and the generators. The paths must be absolute as they are used as bind mount sources.

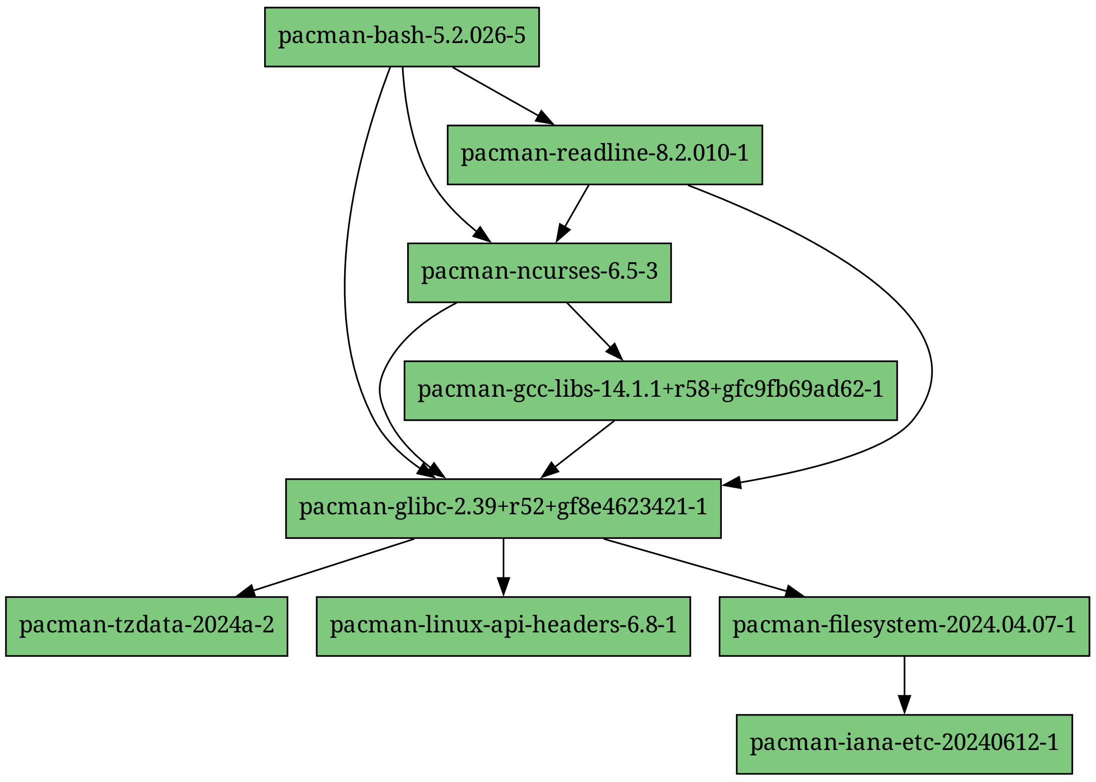
pacman-bash-5.2.026-5
pacman-readline-8.2.010-1
pacman-ncurses-6.5-3
pacman-gcc-libs-14.1.1+r58+gfc9fb69ad62-1
pacman-glibc-2.39+r52+gf8e4623421-1
pacman-tzdata-2024a-2
pacman-linux-api-headers-6.8-1
pacman-filesystem-2024.04.07-1
pacman-iana-etc-20240612-1

Figure A.1 Dependency graph for `examples/input/sh.json`

```
$ ./examples/project/compressor/build-in-container.sh podman \
    "$PWD/tmp/root" "$PWD/tmp/output/generators"
```

After the project is built, we can run the executable. It is a showcase example from a tutorial, so it only outputs a simple message.

```
$ ./examples/project/compressor/build/compressor
```

Dependency Graphs

The meta-manager can produce DOT files describing created dependency graphs during the invocation. After the example script is finished, the graph file is located at `$PWD/tmp/output/graph.dot`.

The DOT file needs to be compiled before viewing. For example, you can use this command from the graphviz software package:

```
$ dot -T pdf tmp/output/graph.dot >tmp/output/graph.pdf
```

To view the graph, you can use any PDF viewer, for example, Firefox:

```
$ firefox tmp/output/graph.pdf
```

Figure A.1 shows an example dependency graph produced by the meta-manager. The input used was the `examples/input/sh.json`. Note that package versions reflect the time at which we ran the command.

A.2.6 Possible Issues

`XDG_RUNTIME_DIR`

The meta-manager example configurations assume that the Podman daemon socket is located at `$XDG_RUNTIME_DIR/podman/podman.sock`. You need to change the path in the compose file if you do not have the `XDG_RUNTIME_DIR` variable set, as in Windows WSL. The file's location is `examples/metamanager/containerized/compose.yaml`.

`WARNING`s

Sometimes, the update or the meta-manager phase output `WARNING` messages. These indicate minor errors when handling old Conan packages. A small number of these messages is not a bug in our application.

Update Errors

The pacman and AUR repositories sometimes spontaneously fail to update. This has always been resolved after deleting the `.PPpackage` directory and waiting a few minutes.

Broken Pipe or Cannot Attach Errors

Try switching to the systemd way of running the Podman daemon if you get container attach errors during package fetching.

A.3 Testing

We used simple regression testing to help detect bugs early. The project is hosted on GitHub, where CI workflows are provided. We created workflows that build the images for containerized invocation and try running the meta-manager on various inputs.

We deem this form of testing a success for this project as the majority of bugs were discovered before merging to the main repository branch.