APPLYING PACKAGE MANAGEMENT TO MOD INSTALLATION

A Thesis
by
STEPHEN BUNN

Submitted to the Graduate School
at Appalachian State University
in partial fulfillment of the requirements for the degree of
MASTER OF SCIENCE

August 2017
Department of Computer Science

APPLYING PACKAGE MANAGEMENT TO MOD INSTALLATION

A Thesis
by
STEPHEN BUNN
August 2017

APPROVED BY:

R. Mitchell Parry, Ph.D.
Chairperson, Thesis Committee

James B. Fenwick Jr., Ph.D.
Member, Thesis Committee

Alice McRae, Ph.D.
Member, Thesis Committee

Rahman Tashakkori, Ph.D.
Member, Thesis Committee

Rahman Tashakkori, Ph.D.
Chairperson, Department of Computer Science

Max C. Poole, Ph.D.
Dean, Cratis D. Williams School of Graduate Studies

Abstract

APPLYING PACKAGE MANAGEMENT TO MOD INSTALLATION

Stephen Bunn
B.S., Appalachian State University
M.S., Appalachian State University

Chairperson: R. Mitchell Parry, Ph.D.

Package management automates the discovery and installation of software that can co-exist within an operating system. The methods used by package management can also address other instances where the installation of software needs to be automated. One example of this is the environment produced by third party video game modifications. An adapted application of package management practices can help to solve the difficult problem of finding and installing a set of video game modifications that do not conflict with each other. This greatly benefits the environment by allowing third party contributions to be easily installed which fosters growth in both the developer and user community surrounding the environment. This thesis presents the theory and complexities behind package management and shows how it can be effectively applied to managing video game modifications by presenting examples of software that can extract relevant metadata from video game modifications and discover conflict free installation solutions.

Acknowledgements

Thanks to Dr. Parry for his insight into what areas of this thesis should be expounded upon and what areas should be cut down so as to tailor to someone new to the idea of package management. Thanks also to my readers, Dr. Tashakkori, Dr. McRae, and Dr. Fenwick for their willingness to read and provide feedback on the content of this thesis. I would like to also like to thank all of the authors of the research papers, technical reports, and documentation that helped me understand package management at a greater depth (beneficial works listed in the bibliography). Thanks also to my parents for their encouragement during the time it took to write this thesis.

Contents

List of Figures

List of Tables

Chapter 1 - Introduction

With the increasing demand for speedy software development, the luxury of not having to *reinvent the wheel* by utilizing third party code has become a staple in all forms of modern software development. In Windows and MacOS systems, the practice of utilizing someone else's code is disguised by bundling all the necessary resources into one setup or archive file that the end user installs. However, in open source systems, mainly Linux, software is typically delivered as either source code or compiled binaries files without including all the necessary third-party resources required to make the software work. In order for the software to function correctly, all of the code that was not bundled with the distributed binaries or source code needs to be separately installed. This can be done manually by searching for, downloading, and installing each required third-party resource. That process can be incredibly inefficient and tiresome for every piece of software one might require. Fortunately, the process of manually installing external resources is automated with a practice termed *package management*.

This method of fitting many small pieces together to form functional software has increased development speed and made the average developer's and Linux user's life easier. Although it is wonderful that this methodology of software delivery has been created, understanding how it works and where it might fit in other areas of development is important for the future of software distribution. Video game modifications is an area

that can greatly benefit from the practices behind package management. Unfortunately, most of the current methods of managing video game modifications do not utilize these practices. By showing that package management can be applied to the distribution of *mods* (packages for video games), this thesis aims to show how package management can be leveraged effectively in other forms of software delivery. The following sections in this thesis describes some of the generic practices and theories behind package management which includes an application of package theory that currently does not exist.

Before continuing into the analysis of a basic architecture involved in package management systems, it is beneficial to understand the projects and communities which have helped in the past to further the quality and completeness of package management systems.

The EDOS (Environment for the Distribution of Open Source software) Project is a European research project with the goal to "improve the stability of a distribution from the point of view of the distribution editor, and not the stability of a particular user installation..." [1]. Much of the reserach done by the collaborators of the EDOS project overviews the organization and issues behind automating the installation of software. Over time the documentation surrounding the origins of the EDOS project has somewhat disappeared. However, the reports and research published by the EDOS team still exists and provides valuable content regarding the challenges and complexity surrounding package management.

The Mancoosi (Managing the Complexity of Open Source Software[1]) project is another European research project that began on early in 2008, and lasted until 2011. Consisting of 37 collaborators within 10 universities and companies, this European research project's tag-line is "managing software complexity" and the problem addressed is the following [2]: "Free and Open Source Software distributions raise difficult problems both for distribution editors and system administrators. Distributions evolve rapidly by integrating new versions of software packages that are independently developed. System upgrades may proceed on different paths depending on the current state of a system and the available software packages, and system administrators are faced with choices of upgrade paths and possibly with failing upgrades." This project is the successor of the previously mentioned EDOS project and provides plenty of beneficial research for optimizing the current state of popular package managers. Many of the researchers who were part of the Mancoosi team have provided research and reports which are used as the foundation in this thesis [3; 4; 5; 6]. Apart from the EDOS project described above, this project intends to help developing tools for the system administrators who are most likely to be interacting heavily with package managers on a day to day basis.

[1] `http://www.dicosmo.org/space/PlaquetteMancoosi-rotated.pdf`

Chapter 2 - What is Package Management?

Package management is the automated installation of software bundles onto a computer system. Unfortunately, the process of performing this automated installation can be quite complex, involving packages, their dependencies and conflicts. To understand package management, we start with the idea of a software *component*.

2.1 Software Components

Components, in the context of software management, are typically viewed as small bundles of software that address a specific need or have a specific purpose. For example, if a developer develops programs in a language that does not have the native ability to predict mimetypes for files, the developer could simply use the open sourced `libmagic` library to handle that responsibility. In this example, the `libmagic` library is a component with the sole responsibility of predicting file mimetypes. Because of this, developers do not typically need to worry about writing mundane libraries for small tasks and instead can focus on real product development.

This idea of components comes from a development methodology called component based software engineering (CBSE). This development methodology isolates concerns into separate components each with a specific purpose.

2.2 Levels of Component Management

Because of the complexity behind free and open source repositories and installers related to Linux distributions, much research has been done regarding these systems. The work done in [7] defines three categories of *component types* that are listed below in order from least to most coarse:

1. *Fine grained components*: Command line tools such as `cat` and `awk` are used and assembled in more complex components using pipes.
2. *Plugins*: Software components which are able to extend the functionality of a particular software. "Plugins are often known as modules or extensions" [7] deployed in the context of an application. Examples include Chrome extensions, Firefox add-ons, source modules for open sourced applications such as the Apache HTTP Server, and Eclipse plugins.
3. *Packages*: An archive which provides either compiled or source code software to some environment (typically the operating system).

According to [7] the more coarse grained a component is, the harder it is to handle. Because of this, packages are typically the most difficult to deal with and therefore are researched heavily.

Before covering some of the more complex features of package management, it is important to define and understand what these packages are and how they relate to each other.

2.3 What is a Package?

Packages are generally defined as an archive file containing a computer program as well as any necessary metadata for its deployment. "This program may either take the form of source code which needs to be automatically compiled or precompiled binaries" [8]. In either case the package provides some set of new features to its level of management. This thesis will use the term package assuming packages are built for free and open source software (FOSS) Debian (DEB) distributions. In order to specify the libraries or other packages that are required for the package to function correctly, package authors use metadata to define *dependencies* on other packages. The definition of dependencies can contain conjunctive and/or disjunctive dependencies. These are typically defined as a list of tuples where disjunctive dependencies are represented as a tuple with two or more packages and a conjunctive dependency is represented with a tuple containing a single package. However, sometimes a package may also be written in a way which breaks the functionality of another. In this case, the package in question lists *conflicts* with other packages. With these two relationships, *depends* (dependencies) and *conflicts* (conflicts), a graph can be formed.

2.4 Relationships between Packages

The graph previously mentioned is often termed a *repository*. This graph is fairly simple using packages as nodes and *depends* and *conflicts* relationships as edges between the nodes. Take the following simple repositories as an example:

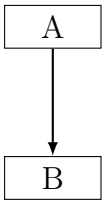


Figure 2.1: Package Dependency

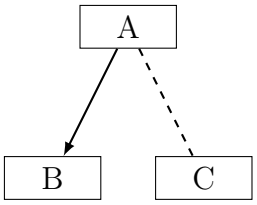


Figure 2.2: Package Conflict

As shown in Figure 2.1, a package A depends on another package B. What this means is that in the repository represented by Figure 2.1, package A must have package B in order to function correctly. In Figure 2.2, however, package A depends on package B but also conflicts with package C. This means that package A must have package B but not package C in order to function correctly.

2.5 Debian Package Management

Previously packages were defined as archives containing either source code or compiled binaries along with descriptive metadata. This section describes the structure and organization of a real world package management system to provide a concrete example of

large scale package distributions. Other distributions such as RedHat Package Manager (RPM) also employ package management, we focus on Debian for simplicity.

2.5.1 Metadata

Debian packages are managed by the `dpkg` package manager and store most of their metadata in a simple text file cal‘led the control file. An example of the popular `tar` package is shown below:

Listing 2.1: Debian Tar Control File

```
Package: tar
Version: 1.28-2.1ubuntu0.1
Architecture: amd64
Essential: yes
Maintainer: Ubuntu Developers <ubuntu-devel-discuss@lists.ubuntu.com>
Installed-Size: 804
Pre-Depends: libacl1 (>= 2.2.51-8), libc6 (>= 2.17), libselinux1 (>= 1.32)
Suggests: bzip2, ncompress, xz-utils, tar-scripts
Conflicts: cpio (<= 2.4.2-38)
Breaks: dpkg-dev (<< 1.14.26)
Replaces: cpio (<< 2.4.2-39)
Section: utils
Priority: required
Multi-Arch: foreign
Description: GNU version of the tar archiving utility
 Tar is a program for packaging a set of files as a single archive in tar
 format.  The function it performs is conceptually similar to cpio, and to
 things like PKZIP in the DOS world.  It is heavily used by the Debian package
 management system, and is useful for performing system backups and exchanging
 sets of files with others.
Original-Maintainer: Bdale Garbee <bdale@gag.com>
```

This file contains a sequence of predefined keys and custom values. A subset of the most useful keys within the context of this thesis is listed below:

- *Package*: The name of the package.
- *Version*: The package's version number.

- *Depends*: If the package depends on any other packages in order to run, a list of package names with specific versions, if necessary, in parentheses. Packages listed here can be installed in any order before or after installation of the parent package (the package the metadata is for).

- *Pre-depends*: Similar to depends but all packages listed under the pre-depends key must be fully installed onto the system before installation of the parent package can be installed.

- *Recommends*: Similar to depends but the listed packages are not necessary for the parent package to function.

- *Suggests*: Similar to depends but the parent package can function without any of the listed packages.

- *Enhances*: The opposite of suggests. Saying "package A enhances package B" is like saying "package B suggests package A."

- *Conflicts*: The opposite of depends. A package cannot be installed if one of the listed conflicts is present on the system.

- *Replaces*: Specifies priority for conflicted packages. If the parent package P conflicts with some package p, and P replaces p, then p will be removed so as to install P.

- *Breaks*: Specifies packages which need to be deconfigured before the package being installed can be unpacked.

- *Provides*: Used to specify what virtual package the parent package falls under (discussed in section 2.5.3).

These packages and their defined relationships can be translated to a partial dependency tree shown in Figure 2.3 which is generated by the `debtree`[1] dependency visualization tool and `graphviz`[2] for graph visualization.

This is just a simple example of a packages dependency tree as the tar package has only three defined dependencies in the Debian distribution. For packages with many more defined dependencies, such as the Firefox web browser, the dependency tree is very extensive.

2.5.2 Archive

Debian packages are distributed as `ar` archives which contain both the metadata and the software the package provides. The software provided can be either source files which need to be compiled or binaries which just need to be moved to a specific location. This section provides a brief overview of these archives so as to present a more in depth understanding of how packages are physically installed to a system.

[1] `https://collab-maint.alioth.debian.org/debtree/`
[2] `http://www.graphviz.org`

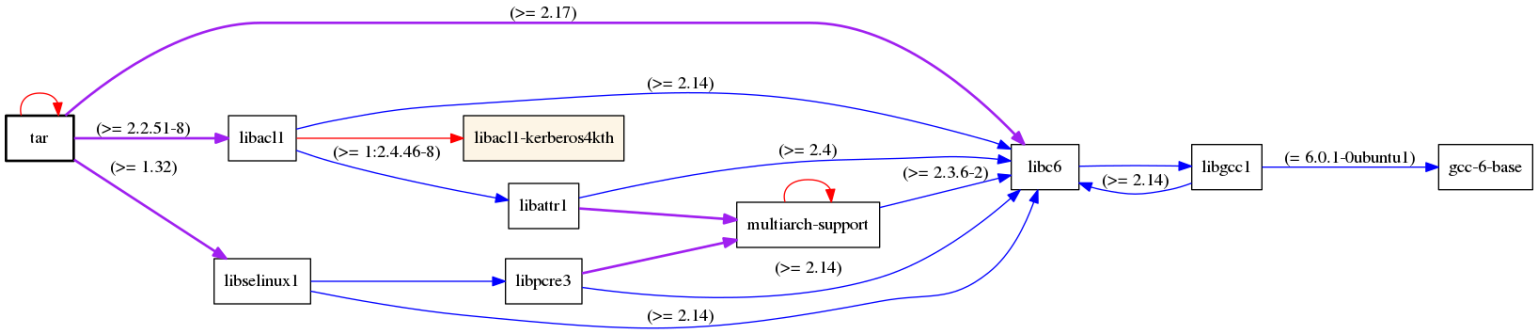


Figure 2.3: Debian Tar Dependency Tree

In the graph shown in the following figure, these edge conventions exist:

- Pre-Depends: purple, bold
- Depends: blue
- Conflicts: red

Debian packages typically are stored as `ar` archives; this means they can be extracted and their contents can be viewed. Using the `ar vx <DEB Package>` command will extract the contents of the archive to the current directory. Debian packages follow a specific folder structure as shown below:

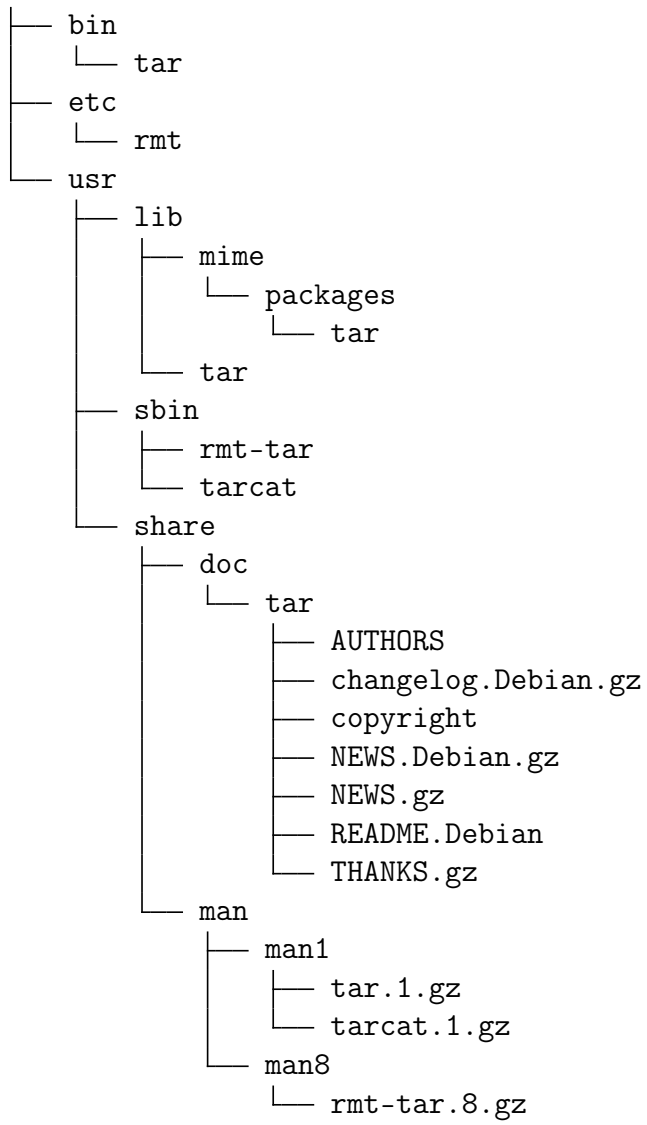
```
├── debian-binary
├── control.tar.gz
│   ├── conffiles
│   ├── prerm
│   ├── postrm
│   ├── control
│   ├── postinst
│   └── md5sums
└── data.tar.xz
```

Notice the control file which was mentioned in section 2.5.1 and shown in Figure 2.3. The nested control archive is where the installer will go to learn how to install the binaries or source available in the data archive. The other file, `debian-binary`, contains the minimum version of Debian required.

Within the control archive several scripts exist. The `postinst`, `prerm`, and `postrm` files contain scripts which are performed post-installation, pre-removal, and post-removal respectively. These scripts are optional and usually used for purging temporary or meta information that has been generated by the package but will no longer be needed after removal. The `conffiles` file is a file which is used by the package manager to help the user maintain their configuration of a package after an upgrade. When packages are upgraded, they can potentially remove user configuration through the `prerm` or `postrm`

scripts. The `conffiles` file is meant to ensure user configuration persists. Finally, `md5sums` contains the checksums of files stored within the data archive for validation and build checking before installation.

The data archive varies from package to package but consists of binaries which are to be installed onto the system. These binaries are nested within sub-directories that indicate where on the system these binaries should be placed. An example of an extracted `tar` data archive can be seen below:



2.5.3 Virtual Packages

Virtual packages exist in every modern form of package management and are very useful for describing the basic function of a package. Typically represented by the *Provides* metadata tag, the value is an identifier recognized by the system as a sort of application

group. A simple example is that the `firefox` package provides the `web-browser` identifier. Many other packages can also provide this identifier such as the package for the Chrome browser.

This identifier becomes useful when packages require a web browser, so they list `web-browser` as a requirement in their respective dependency lists. This feature allows for either the installation of the Firefox or the Chrome browser to satisfy this dependency. This requirement is "considered as a disjunctive requirement whose elements are all packages that provide web-browser" [4].

2.6 Installers

The program that uses the metadata to determine and install packages required for a specified package to work is called the installer. The installer is actually split into two different steps called the meta-installer and the physical-installer. The purpose and inner workings of these installers is documented in the following subsections.

2.6.1 Meta Installers

In order to correctly install a package onto a system, the package manager must discern which dependencies to install given the listed conflicts and the already present packages on the system. This process is termed dependency solving [9] and is discussed in depth as the most important process of package management. Ultimately, the meta-installer actually

handles the resolution of dependencies for a given package [6]. Much of the research in package management focuses on the implementation of meta-installers, particularly the algorithms behind dependency solving and the issues they face [6]. The following subsections provide an overview of the research that addresses solutions to dependencies. Since more research has been done regarding Debian meta-installers, this section will consider only the following meta-installers built for Debian distributions:

- `apt`: The Advanced Packaging Tool originally built as a front end to Debian's `dpkg` package utility for `.deb` packages. Although built for Debian, it has been adapted to RPM packages under the `apt-rpm` package.
- `aptitude`: An Ncurses based front-end to the `apt` utility for Debian packages.
- `smart`: A full-fledged meta-installer built in Python with included heuristics to help with solving complex dependency solving.
- `apt-pbo`: A meta-installation tool proposed by [9] aimed to improve dependency solving using pseudo-boolean optimization (PBO). Adds the idea of installation policies such as freshness, removal, and number.

2.6.1.1 Dependency Solving

A responsibility of the meta-installer is to discover possible solutions to an upgrade problem. This process is known as dependency solving, and it occurs in every form of

package management. As shown in chapter 3.2 and proven in [7], dependency solving is NP-complete. As discussed in [9], "Finding a solution becomes rapidly more difficult as the number of available packages grows and the number of versions of each package increases." This presents many issues as many distribution's repositories are quite large and provide many versions to multiple packages. Typically solvers used by these package managers are prebuilt SAT solvers which do the job quickly and provide a valid response. The solver does this by encoding a package's dependency tree as a conjunctive normal form (CNF) satisfiability problem using several simple steps shown in section 5.3.

Although many dependency solvers are based off of SAT, "it should be noted that SAT is a decision problem and any solution to the problem is equally valid" [9]. In order to combat the issue where some solutions are better than others, "an optimization problem and a extension of the SAT formulation called pseudo-Boolean optimization (PBO) should be used instead" [9]. However, effective use of PBO is not standard and is slower than the use of an SAT solver, therefore the rest of this thesis will focus mainly on use of SAT solvers rather than PBO solvers.

2.6.2 Physical Installers

The physical-installer is rather simple in comparison to the meta-installer. This installer simply takes the downloaded packages as reported by the meta-installer, extracts, and then builds and/or places them on the system for use by the user. "Downloading packages

and resolving dependencies between packages are in general beyond the scope of the installer" [10]. Typically these installers are specific to the type of system they are installing the package on. For example, Debian utilizes the `dpkg` installer for installing packages onto a Debian based system.

Chapter 3 - Package Theory

In order to understand how to determine whether a package can be installed on an existing system, it is necessary to formally define installability and the relationships surrounding packages. This following sections provide details on how this task is completed by using the generic definitions compiled by Jaap Boender using the formalization of *package theory* [3; 11; 4]. Some of these definitions are described and adapted below in order to accurately describe package installability.

3.1 Definitions

Definition 3.1.1 (Repository) *A repository* $R = (P, D, C)$ *is a triple consisting of a set of packages* P*, a conflict relation* C*,* $(C \subseteq P \times P)$*, and a dependency function* $D : P \longrightarrow\subseteq \mathscr{P}(P)$*, where* $\mathscr{P}(\cdot)$ *is the power set operation* [3].

This simply states that a repository has a set of packages where conflicts and dependencies may exist. Given this definition, it is beneficial to also take into consideration the following example of a repository [3]:

$$
\begin{aligned}
P &= \{\text{alpha, bravo, charlie, delta, epsilon, foxtrot}\} \\
D(\text{alpha}) &= \{\{\text{bravo}\}, \{\text{charlie, delta}\}\} \\
D(\text{bravo}) &= \{\} \\
D(\text{charlie}) &= \{\} \\
D(\text{delta}) &= \{\} \\
D(\text{epsilon}) &= \{\{\text{delta, foxtrot}\}\} \\
D(\text{foxtrot}) &= \{\} \\
C &= \{(\text{delta, foxtrot}), (\text{foxtrot, delta})\}
\end{aligned}
$$

As shown above, a package p cannot depend or conflict with itself. It is also interesting to note that if the conflict (c_1, c_2) exists, then the conflict (c_2, c_1) also exists. These properties are formally described in two axioms used in common FOSS package management relating to packages:

Axiom 1 (Package Self Dependency) *For any package $p \in P$, there wouldn't be a $d \in D(p)$ such that $p \in d$ (remember that d is actually a set)* [3].

Axiom 2 (Package Self Confliction) *The conflict relation follows $\forall_{(p_1,p_2)\in C}\ p_1 \neq p_2$ and $\forall_{(p_1,p_2)\in C}\ (p_2, p_1) \in C$ (symmetric and irreflexive)* [3].

These two axioms ensure that a package cannot depend on or conflict with itself.

"For a package p to be installable with respect to a given repository (P, D, C), it must be possible to find a set I of packages in the repository (the *installation*; $I \subseteq P$) that contains the package and fulfills two conditions; all the dependencies in I must be

satisfied and no two packages in I are in conflict" [3] with each other. This definition of installability can be decomposed into two properties: *abundance* and *peace*.

Definition 3.1.2 (Abundance) *A set of packages I is abundant (with respect to a repository (P, D, C)) if and only if* $\forall_{p \in I}[\forall_{d \in D(p)}[I \cap d \neq \emptyset]]$ [3].

This means that an installation set satisfies all the dependencies of all its packages.

Definition 3.1.3 (Peace) *A set of packages I is peaceful (with respect to a repository (P, D, C)) if and only if* $\forall_{(c_1,c_2) \in C}[\neg(c_1 \in I \wedge c_2 \in I)]$ [3].

This definition simply states that a set of packages is peaceful with respect to a repository if there are no conflicts between two packages in that set. Boender continues to show that the combination of these two terms leads to another property referred to as *health*.

Definition 3.1.4 (Health) *A set of packages is healthy with respect to a repository R if it is abundant and peaceful with respect to R* [3].

With the previous definitions, it is possible to formally define the actual problem behind package management which is the *installability* of a package.

Definition 3.1.5 (Installability) *A package p is installable in a repository $R = (P, D, C)$ if and only if there exists a healthy set $I \subseteq P$ such that $p \in I$* [3].

There can be multiple installation sets for a package p, which are called the *installation sets* of p. A simple extension to the idea of installability is that if two or more packages are installable at the same time for a specific repository, they are co-installable.

Definition 3.1.6 (Co-Installability) *A set of packages S is co-installable in a repository $R = (P, D, C)$ if and only if there exists a healthy set $I \subseteq P$ such that $S \subseteq I$* [3].

Now that package installability has been formally defined, formal definitions for the relationships between packages must be also be defined. It is important to differentiate between direct vs. indirect dependencies and disjunctive vs. conjunctive dependencies.

Definition 3.1.7 (Direct Dependency) *A package p depends on another package q, $(p \underset{1}{\rightarrow} q)$ if and only if there is a $d \in D(p)$ such that $q \in d$* [3].

This means that a package p depends directly on any package that it specifically lists in one of its dependencies, d. Because multiple packages can satisfy the same dependency such as $\{\text{charlie}, \text{delta}\}$ above, not every package dependency is required. With this dependency relation, it is now possible to form a graph (V, E) for a repository (P, D, C) where $V = P$ and $E = \{(p, q) \big| p \rightarrow q\}$ [3].

Definition 3.1.8 (Dependency) *A package p depends on another package q, $(p \rightarrow q)$ if and only if there is a list of packages $p_1, p_2, \cdots, p_n$ such that $p \rightarrow p_1 \rightarrow p_2 \rightarrow \cdots \rightarrow p_n \rightarrow q$* [3].

The general term dependency refers to a package q which may be required by a package p to function although package q may not be a direct dependency of package p. Again, not all of these packages are required because multiple packages can satisfy the same dependency.

This can be visualized as a package that is accessible from any path starting from a package p.

Definition 3.1.9 (Conjunctive Direct Dependency) *A package p has a conjunctive direct dependency on a package q, ($p \xrightarrow[1]{c} q$) if and only if there is a $d \in D(p)$ such that $d = \{q\}$* [3].

A conjunctive direct dependency is one where a package p depends on package q and cannot be installed without it. A conjunctive dependency simply extends this idea to a sequence of conjunctive direct dependencies:

Definition 3.1.10 (Conjunctive Dependency) *A package p is a conjunctive dependency on package q, ($p \xrightarrow{c} q$) if and only if there is a list of packages $p_1, p_2, \cdots, p_n$ such that $p \xrightarrow{c} p_1 \xrightarrow{c} p_2 \xrightarrow{c} \cdots \xrightarrow{c} p_n \xrightarrow{c} q$* [3].

From this definition of conjunctive dependencies, it is easy to see that for two packages p and q such that $p \xrightarrow{c} q$, any installation set of p must include q [3].

Now we consider the set of all packages that a given package depends upon. A *dependency cone* is defined as follows:

Definition 3.1.11 (Dependency Cone) *The dependency cone* $\Delta_R(p)$ *of a package* p *with respect to a repository* $R = (P, D, C)$ *is the set of packages* $\{q \in P \mid p \rightarrow q\}$. *Similarly, the dependency cone* $\Delta_R(S)$ *of a set of packages* S *is the union* $\bigcup_{p \in S} \Delta_R(p)$ [3].

This simply means that the dependency cone of a package p with respect to a repository R is the set of packages that p depends on. Related to the dependency cone is the reverse dependency cone. This cone is the set of packages which depend on a package p.

Definition 3.1.12 (Reverse Dependency Cone) *The reverse dependency cone* $\nabla_R(p)$ *of a package* p *with respect to a repository* $R = (P, D, C)$ *is the set of packages* $\{q \in P \mid q \rightarrow p\}$. *Similarly, the reverse dependency cone* $\nabla_R(S)$ *of a set of packages* S *is the union* $\bigcup_{p \in S} \nabla_R(p)$ [3].

If a package p is needed for a package q to function, then q is in the reverse dependency cone of p, $(\nabla_R(p))$. The definition of dependency cones simplifies the task of determining if a package is installable because it reduces the number of packages to consider:

Definition 3.1.13 (Installability by Dependency Cone) *A package* p *is installable with respect to a repository* $R = (P, D, C)$ *if and only if it is installable with respect to the repository* $(\Delta_R(p), D, C)$ [3].

There are many more useful definitions in Boender's paper [3]; however, they are not related to the research of this thesis.

3.2 Package Installability

The meta-installer (discussed in section 2.6.1) must determine whether or not a package is installable (definition 3.1.5) given a repository. However, it has been proven that determining whether a package is installable is NP-complete due to the fact that "package installation is in NP" and "any 3SAT problem can be reduced in polynomial time to an instance of the package installation problem" [7]. Fortunately, practical problems are still tractable [3] and can be solved as an instance of the Boolean satisfiability problem.

The Boolean satisfiability (SAT) problem's purpose is to find the possible configurations of Boolean variables that makes a given Boolean expression true. For example, the Boolean expression $(\neg A \wedge B)$ is solvable when $A = \text{False}$ and $B = \text{True}$. One of the most popular ways of representing a Boolean expression is by using conjunctive normal form (CNF). This form is a conjunction of clauses with disjunctive literals. For example, the Boolean expression $(A \vee \neg B \vee \neg C) \wedge (A \vee B \vee \neg C)$ is an instance of CNF because each literal (A, B, C) are disjunctive within each clause, and each clause is conjunctive within the expression.

Determining whether package p is installable within a repository $R = (P, D, C)$ (dependency solving) can be done by encoding the problem as a CNF expression. The expression can be built using the following steps [9]:

1. Add the clause (p)
2. Add the clauses $\left\{ \left(\neg q \vee \left(\bigvee_{d \in D(q)} \{i \in d\} \right) \right) \;\middle|\; p = q \vee p \rightarrow q \right\}$
3. Add the clauses $\{(\neg c_1 \vee \neg c_2) \mid (c_1, c_2) \in C,\ (p = c_1 \vee p \rightarrow c_1) \wedge (p = c_2 \vee p \rightarrow c_2)\}$
4. These clauses are combined using `AND`

Using these steps, one can encode the potential installation of a package p into a Boolean expression. Take for example the following repository graph:

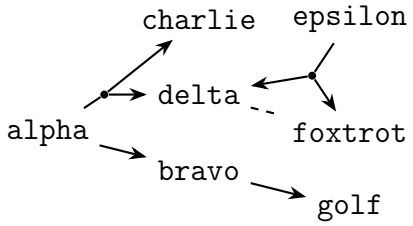


Figure 3.1: Simple Repository Graph

The installation of `alpha` for the graph shown above can be encoded as the following Boolean expression:

$$(\texttt{alpha}) \wedge (\neg\texttt{alpha} \vee \texttt{bravo}) \wedge (\neg\texttt{alpha} \vee \texttt{charlie} \vee \texttt{delta}) \wedge (\neg\texttt{bravo} \vee \texttt{golf})$$

The clauses which result from conflicts can be seen in the encoding of the installation of `epsilon`:

$$(\texttt{epsilon}) \wedge (\neg\texttt{epsilon} \vee \texttt{delta} \vee \texttt{foxtrot}) \wedge (\neg\texttt{delta} \vee \neg\texttt{foxtrot})$$

This encoding can ultimately be used by an SAT solver to discover possible installation solutions. The complexity of solving the CNF expression is dependent on the number of literals within the expression which need to be considered. In the instance shown above, the number of unique literals is the number of packages within $\Delta_R(p)$ of a package p. Because a package must either be true (installed) or false (uninstalled) within a solution, $2^{|\Delta_R(p)|}$ different configurations of packages could be exhaustively evaluated.

The number of packages which need to be considered when determining installability can be further reduced by removing superfluous dependencies in the dependency cone of p. This can be done by removing all dependencies for $q \in \Delta_R(p)$ for $d \in D(q)$ where $\exists_{d' \in D(q)}[d \subset d']$ is true [3]. Doing this removes unnecessary dependencies from a healthy set of packages and decreases the number of literals that need to be considered when determining package installability. By reducing the number of package relationships which need to be considered by a solver, installability problems remain to be tractable [3].

Chapter 4 - Video Game Modifications

A video game modification, typically shortened to *mod*, is an extension which changes the execution of the game in some way. Mods typically add assets or logic to the game. For example, assets might include textures, 3D models, or sounds. Logic determines how the game uses the assets. For the purpose of this thesis, we will use the following definition for mod:

The following sections describe the practical complexity of installing mods for games published by Bethesda[1], and how mod installability relates to package theory.

4.1 Installing Mods

For Bethesda games, mod installation occurs dynamically at runtime in a particular *load order*[2]. For example, one of Bethesda's more popular games, Skyrim, has standard game resources which are loaded when the game starts. If the game is executed with these resources, the game understands when, where, and how to render them. However, Bethesda has released additional downloadable content (DLC) which needs its own resources (apart from the standard ones). In order to do this, the DLC contains new files that can be placed within the game's directory structure so they can be accessed by the game. However, the game needs to know what resources exist.

[1] `https://bethesda.net`

[2] `https://bethesda.net/community/topic/5795/how-to-load-order`

To understand how the game discovers the additional resources, it is necessary to know that a Bethesda based game has two main types of resources:

- *Asset*: textures, 3D models, sounds, etc.
- *Logic*: modifies how the game functions

Assets typically remain on the file system while new logic is loaded at runtime.

In earlier versions of Bethesda games, the load order was determined by the timestamps of the logic resources being added. However, more recent versions of Bethesda games utilize a text file called `plugins.txt` to indicate what plugins (up to 255) should be loaded and in what order. Once loaded by the game, the logic refers to the assets within the game directory.

Within the scope of DLC that is officially published by Bethesda, there are no instances where adding the DLC overwrites or breaks the execution of the game. However, when user-written DLC (i.e., mods) are loaded into the game, logic or assets may conflict with previously loaded mods. These conflicts require an analysis to determine whether a mod is installable. Mod installability is discussed in the following sections along with a more in-depth analysis of mod conflicts and dependency relationships.

4.2 Installability Analysis

As mentioned earlier, package installability is NP-complete. *Mod Installability* is an instance of package installability which has its own conflicts and dependencies.

4.2.1 Mod Conflicts

As opposed to package management, mod management does not have very many instances of conflicts which break execution of the game. Instead, a mod has conflicts which hinder user experience. This is due to the previously mentioned load order which only affects logic additions to the game. Any mod that has some assets already defined may be intentionally or unintentionally overwritten by the loading of a mod later in the load order. In most cases, only noncritical assets are overwritten and thus do not cause the game to crash. However, it may cause instances where missing or broken assets are loaded into the game thus breaking the player's immersion. This is not the only form of conflict possible within Bethesda-based mods. Specifically, we define three categories of conflicts that a mod may have with another mod. These conflict categories are listed below in order from the most to the least common:

1. *Asset File Conflicts*: These are the most common form of conflicts and occur in mods that write assets into the games directory structure. These files could

overwrite already existing asset files. File conflicts are detected by checking the path for each added file against previously existing file paths.

2. *Asset Reference Conflicts*: Within the game, assets are reference by links and mod logic additions can change these links. This can create a conflict when a previously loaded mod requires a specific asset reference and a mod loaded later changes the reference. Asset conflicts are detected by checking the references for each added file against previously existing references.

3. *Logic Behavior Conflicts*: In addition to specifying the asset references, logic also defines the behaviors of objects within the game. When a logic addition contradicts the logic of a previously loaded mod, this causes a conflict. For example, a previous mod may determine that a player has died, a new mod could bring that player back to life. Detecting logic behavior conflicts is outside the scope of this thesis and a good topic for future work.

4.2.2 Mod Dependencies

With respect to mod load ordering, there are situations where a mod p that requires a mod q to be loaded by the game before mod p can be loaded properly defines a dependency $p \rightarrow q$. In Bethesda-based mods, a logic asset contains some references to other logic assets that must be loaded by the game first. This relationship is similar to a direct dependency in package theory. For example, the popular mod Frostfall references the

following logic assets:

$$D(\texttt{Frostfall.esp}) = \{\{\texttt{Skyrim.esm}\}, \{\texttt{Update.esm}\}, \{\texttt{Campfire.esm}\}\}$$

These dependencies are commonly called *masters* in Bethesda mods. If a mod $\texttt{Frostfall} \underset{1}{\rightarrow} \texttt{Campfire}$ then `Campfire` is a master of `Frostfall`. From the dependency function shown above, one can clearly see that the file `Frostfall.esp` has three explicitly declared dependencies. These are the standard game `Skyrim.esm`, the published unofficial update to the standard game `Update.esm`, and another mod `Campfire.esm`. Because some of these declared dependencies also have their own dependencies, a dependency tree can be built from these relationships. One can further develop the dependency tree by including the declared dependencies of `Campfire.esm` as shown below:

$$D(\texttt{Campfire.esm}) = \{\{\texttt{Skyrim.esm}\}, \{\texttt{Update.esm}\}\}$$

As shown above, `Campfire.esm` only depends on two standard assets already in the dependency tree. Therefore, this dependency tree is complete as both `Update.esm` depends on `Skyrim.esm` and `Skyrim.esm` depends on nothing. Dependencies such as `Skyrim.esm` in Bethesda-based mod management are analogous to dependencies such as `glibc` shown in Figure 2.3.

There are dependencies that do not need to overwrite previous logic and therefore do not have dependencies. One example is the popular `SKSE` (*Skyrim Script Extender*) extension that takes the form of a *dynamic-link library* rather than the previously discussed formats. This extension mainly adds many new functions and frameworks for mod developers to utilize during development. For many mods, `SKSE` is a required dependency, where the mod will not function unless `SKSE` is dynamically loaded by the game as well. This thesis assumes that all required dynamic-link libraries are loaded at runtime. With this knowledge, one can define two different types of dependencies that currently exist in Bethesda-based mod management.

- *Explicitly Defined Dependencies*: Dependencies which are typically explicitly defined as masters within the header of a TES file. Explicitly defined dependencies are detected as described above.

- *Implicitly Defined Dependencies*: Dependencies which are not explicitly defined as masters or dependencies but are nonetheless needed for proper execution of the mod. Dynamic-link libraries are a good example of implicit dependencies. These dependencies are detected by manually reading mod author documentation.

Because dependencies in Bethesda based mods initially appear to be conflicts, if a conflict between a mod and its master is detected, it can be safely assumed that the master's resources being overwritten is expected.

Chapter 5 - Applied Mod Management

Now that the loading, dependencies, and conflicts of Bethesda based mods have been described, the following sections provide details on how to extract this information from the mods themselves and how to to find installable solutions. Valid installation solutions are considered to be any solution where obvious conflicts do not cause the player's immersion to break.

5.1 Mod Structure

The following sections describe the basic structure that Bethesda based mods employ. This structure was reverse engineered as part of this thesis with help from the community posts at the Unofficial Elder Scrolls Pages[1].

Bethesda's game engine is specifically built to read logic from *esm* and *esp* files using the TES (The Elder Scrolls) file format. There are several versions of TES file formats. The most popular is the TES4 (TES version 4) file format although a newer version (TES5) also exists. This thesis focuses on TES4 files because far more exist. Assets are loaded as either a simple folder hierarchy (typically termed *loose files*) or more recently as an archived *BSA* (Bethesda Archive) or *BA2* (Bethesda Archive 2) file. This thesis reverse engineers the BSA files because the vast majority of TES4 files

[1] `http://en.uesp.net`

reference only them. The following subsections detail the general structure of the TES files and the BSA archive reverse engineered for this thesis.

5.1.1 TES (The Elder Scrolls) File Format

TES files are those previously termed *ESM* and *ESP*. Both of these file types follow the same format except for some minor differences.

A TES4 file contains three different data model types, *groups*, *records*, and *fields*. A given TES4 file begins with a header which takes the form of a record using the following structure:

Table 5.1: TES4 Record Structure

TES4 Record		
Name	**Type**	**Description**
name	`ubyte[4]`	The record's name supported by the engine
size	`ulong`	Size of the content following the heading
flags	`ulong`	Bit flags for 10 unique meanings
form id	`ulong`	Engine reference for the record
vc info	`ubyte[4]`	Various information for version control
version	`ulong`	Record format version number
content	`byte[size]`	Field storage for the record

A record defines a piece of logic to be utilized by the game. The record defines what kind of logic is being added to the game through a combination of the record's name and the fields stored in the record's content. These fields use the following structure:

Table 5.2: TES4 Field Structure

TES4 Field		
Name	**Type**	**Description**
name	`ubyte[4]`	Name of the field from the supported field names
size	`ushort`	Size of the data of the field
data	`byte[size]`	Data the field provides

These fields have a set of names that can be used, which depending on its name and its parent record, has a specific meaning and expected data type.

For example for a TES4 file, the name of the header record will be `TES4` and have a form id of `0`. For example, the below output is an example of a header record and its nested fields for the popular mod Frostfall:

```
<TES4Record (TES4) "0x0">
  <TES4Field ('HEDR') '0x9a99d93f170500007c4a5c74'>
  <TES4Field ('CNAM') 'Chesko'>
  <TES4Field ('SNAM') 'Adds hypothermia and cold weather survival mechanics.'>
  <TES4Field ('MAST') 'Skyrim.esm'>
  <TES4Field ('DATA') '0x0000000000000000'>
  <TES4Field ('MAST') 'Update.esm'>
  <TES4Field ('DATA') '0x0000000000000000'>
  <TES4Field ('MAST') 'Campfire.esm'>
  <TES4Field ('DATA') '0x0000000000000000'>
  <TES4Field ('INTV') '0x01000000'>
```

Listing 5.1: Sample TES4 Header Record

As shown above, the header record has the name `TES4` and a form id of `0x0` (`0`). This record has 10 fields, the notable ones being `HEDR`, `CNAM`, `SNAM`, and `MAST`. The `HEDR` field is a unique field that identifies a specific record as the TES header. `CNAM` and `SNAM` fields simply denote the TES file author and the TES file description, respectively. The other

interesting field is the `MAST` field, a field that explicitly defines masters (dependencies). `DATA` fields are typically used as a delimiter between `MAST` fields.

Following the header record will be a list of groups using the following structure:

Table 5.3: TES4 Group Structure

TES4 Group		
Name	**Type**	**Description**
name	`ubyte[4]`	Always `GRUP`
size	`ulong`	Total size of the group
label	`ubyte[4]`	Record types within this group
group	`long`	Indicates group type from 10 unique groups types
timestamp	`ulong`	Timestamp of the group edit, MSDOS format
version	`ulong`	Group format version number
content	`byte[size-0x18]`	Record storage for the group

These groups are identified by their names and contain records using the same name. For example, the following output is an example of a group and two of that group's records.

```
<TES4Group (TOP) 'LSCR'>
  <TES4Record (LSCR) "0x303df3e">
    <TES4Field ('EDID') '_Frost_LoadingScreen12'>
    <TES4Field ('DESC') 'Being wet causes you to gain exposure much more rapidly.'>
    <TES4Field ('NNAM') '0x3cdf0303'>
    <TES4Field ('SNAM') '0x6666663f'>
    <TES4Field ('RNAM') '0xf6ff0000ddff'>
    <TES4Field ('ONAM') '0xd3ff2d'>
    <TES4Field ('XNAM') '0x000006342000000000000086332'>
    <TES4Field ('MOD2') 'Cameras\\LSCameraPanZoomInSmall.nif'>
  <TES4Record (LSCR) "0x305f404">
    <TES4Field ('EDID') '_Frost_LoadingScreen20'>
    <TES4Field ('DESC') 'Vapor Blast only deals damage if the caster is wet.'>
    <TES4Field ('NNAM') '0x87d11000'>
    <TES4Field ('SNAM') '0x00002040'>
    <TES4Field ('RNAM') '0x0000f1ff0000'>
    <TES4Field ('ONAM') '0x4cffb4'>
    <TES4Field ('XNAM') '0x00005fc100000000000070c100'>
    <TES4Field ('MOD2') 'Cameras\\LSCameraPanZoomInSmall.nif'>
```

Listing 5.2: Sample TES4 Group

The above top-level group (`TOP`) defines some new loading screens in the form of records (only two shown). Within each of the `LSCR` (loading screen) records, there are fields which describe the content of the record. The most notable ones are the `EDID` and the `DESC` fields. The `EDID` is a very important record as it specifies the editor id of the new content being added. In this case the author, Chesko, has named some of the new loading screens `_Frost_LoadingScreen12` and `_Frost_LoadingScreen20`. The `DESC` field within the context of the `LSCR` record defines the text to be shown on the loading screen. It is important to note that back in the `TES4` record (the header record), the `SNAM` field was used to provide a description. However, in this instance the `DESC` field is used for a description and the `SNAM` field has a completely different meaning. This is because records can use the same fields as another record but have different meanings for the data within those fields, adding to the complexity. Therefore, fields cannot be correctly interpreted without knowing the context in which they are being used. The header in combination with multiple groups is what makes up the full content of the TES4 file.

The following is a snippet of code using these structures to extract the explicitly defined masters of the previously mentioned Frostfall mod from the header record:

```
import os
import modage

# gets the appropriate plugin class for the given file, in this case TES4
plugin = modage.plugin.get_plugin('~/Downloads/Frostfall.esp')
print((
    os.path.basename(plugin.filepath), tuple(
        _.content.decode('ascii')[:-1]
        for _ in plugin.header.fields
        if _.name == b'MAST'
    ),
))
```

Listing 5.3: Display Explicit Masters

This results in the following being printed:

```
('Frostfall.esp', ('Skyrim.esm', 'Update.esm', 'Campfire.esm'))
```

Methods similar to this are useful for extracting information from the TES files in order to determine relationships between mods.

5.1.2 BSA (Bethesda Archive) File Format

The TES4 file only dictates logic for the Bethesda engine. In order to add assets and new content, the mod has to provide its assets in either loose file or archived form. One of the most common archive formats is the *BSA* archive. This archive starts with a header using the following structure:

Table 5.4: BSA Header Structure

BSA Header		
Name	**Type**	**Description**
bsa	`ubyte[4]`	BSA header magic, always `BSA`
version	`ulong`	BSA archive version
offset	`ulong`	The offset of the header's end
primary flags	`ulong`	Bit flags, unkown meanings
folder count	`ulong`	Count of folders in the archive
file count	`ulong`	Count of files in the archive
folder names length	`ulong`	Max length of folder names
file names length	`ulong`	Max length of file names
secondary flags	`ulong`	Bit flags, unknown meanings

The Frostfall mod shown in section 5.1.1 also provides a BSA archive. Below is an example of the decoded header for the mod Frostfall.

```
BSAHeader(
  bsa='BSA', version=104, offset=36, primary_flags=31,
  folder_count=9, file_count=271,
  folder_names_length=166, file_names_length=8287,
  secondary_flags=511
)
```

Listing 5.4: Sample BSA Header

From reading this header, one can tell that this BSA clearly contains 9 folders and 271 files. Following this header is a list of folders using the following structure:

Table 5.5: BSA Folder Structure

BSA Folder		
Name	**Type**	**Description**
hash	`ulonglong`	Path hash used for true indexing
file count	`ulong`	The number of files in the folder
offset	`ulong`	The offset the folder's files start at

Shown below are the first three folder structures within Frostfall's BSA.

```
BSAFolder(hash=1275205697802562668, file_count=2, offset=8467)
BSAFolder(hash=1495643880841964652, file_count=21, offset=8520)
BSAFolder(hash=1948419268744733541, file_count=96, offset=8876)
```

Listing 5.5: Three BSA Folder Structures

Following this list of folders is the list of files contained within those folders. These files use a similar structure to the BSA folder:

Table 5.6: BSA File Structure

BSA File		
Name	**Type**	**Description**
hash	`ulonglong`	Path hash used for true indexing
size	`ulong`	The size of the file in bytes
offset	`ulong`	The offset to the file's start

Below are the first three file structures within Frostfall's BSA.

```
BSAFile(
  hash=5548092738682287080, size=7420, offset=12978,
  path='interface/frostfall', name='frostfallskyuisplash.dds'
)
BSAFile(
  hash=13293804818063717224, size=5010, offset=20398,
  path='interface/frostfall', name='frostfall_splash.swf'
)
BSAFile(
  hash=732761691906355377, size=35565, offset=25408,
  path='textures/frostfall', name='_frost_plainglasstile01.dds'
)
```

Listing 5.6: Three BSA File Structures

Although they are not listed in Table 5.6, both the `path` and `name` fields can be extracted by going though the file at a given offset and reading until a null terminator is reached. This archive extracts to a simple folder hierarchy typically containing engine specific assets such as Microsoft DirectDraw Surface (`.dds`) files. This archive is not used for

compression, and there is never an instance where TES files are stored within a BSA archive.

In distribution, these files (TES, BSA, etc.) are typically laid out in an undefined folder hierarchy and compressed with a generic archive format such as *LZMA*, *zip*, *rar*, etc. One of the many issues with proper mod distribution is the lack of a structured folder hierarchy. Without the use of a structured folder hierarchy, it becomes more complex to automate the downloading and installing of the assets required by the mod. More information about this issue and several others that plague current mod distribution practices are discussed in section 6.3.

5.2 Constructing Mod Dependency Cones

As shown in 5.1.1, it is currently possible to extract certain pieces of metadata directly from the mods and archives that are currently being distributed. However, it should be noted that the conflict and dependency relationships that currently can be extracted are not complete, meaning that not all of the conflicts and dependency types listed in section 4.2 are extracted. Specifically the implicit dependencies discussed in section 4.2 are not extracted. In order to perform simple dependency solving with these mods, it requires the ability to (at least naïvely; not completely) extract dependency and conflicts relationships into a distribution of metadata.

Because the type of compression used for mod archiving and folder hierarchy is not predictable, the extraction of metadata was not fully automated and had to be tailored for several mods. Therefore, a subset of mods within the repository was picked to show the effects of dependency solving on mod installation problems that may occur. First, the metadata for the standard game Skyrim including all relevant Bethesda published DLCs was defined as the following dictionary:

```
VANILLA_DISTRIBUTION = {
    "bethesda/skyrim": {
        "versions": {
            "0.0.0": {
                "provides": ["Skyrim.esm"],
                "depends": [],
                "conflicts": [],
            }
        }
    },
    "bethesda/update": {
        "versions": {
            "0.0.0": {
                "provides": ["Update.esm"],
                "depends": [("bethesda/skyrim:latest",)],
                "conflicts": []
            }
        }
    },
    "bethesda/dlc/hearthfire": {
        "versions": {
            "0.0.0": {
                "provides": ["Hearthfire.esm"],
                "depends": [("bethesda/skyrim:latest",)],
                "conflicts": []
            }
        }
    },
    "bethesda/dlc/dawnguard": {
        "versions": {
            "0.0.0": {
                "provides": ["Dawnguard.esm"],
                "depends": [("bethesda/skyrim:latest",)],
                "conflicts": []
            }
        }
    },
    "bethesda/dlc/dragonborn": {
        "versions": {
            "0.0.0": {
                "provides": ["Dragonborn.esm"],
                "depends": [("bethesda/skyrim:latest",)],
                "conflicts": []
            }
        }
    }
}
```

Listing 5.7: Vanilla Distribution Metadata

The distribution of commonly used dynamic libraries was represented by the following:

```
DLL_DISTRIBUTION = {
    "dll/skse": {
        "versions": {
            "0.0.0": {
                "provides": [],
                "depends": [("bethesda/skyrim:latest",)],
                "conflicts": []
            }
        }
    },
    "dll/enb": {
        "versions": {
            "0.0.0": {
                "provides": [],
                "depends": [("bethesda/skyrim:latest",)],
                "conflicts": []
            }
        }
    }
}
```

Listing 5.8: DLL Distribution Metadata

With these two defined, all base packages can be encapsulated into a single game distribution dictionary. Using this structure of metadata, the encoding of user created mods and their relationships can be automated using methods similar to the one mentioned in section 5.1.1. A tailored list of mods was extracted and placed in another dictionary called `MOD_DISTRIBUTION`. The combination of these three dictionaries creates the `FULL_DISTRIBUTION` map. Using this `FULL_DISTRIBUTION` map, we can perform dependency solving using the same practices as package management.

5.3 Dependency Solving

In order to accurately get the information for a specified version of a mod, the following help function was created:

```
def get_specification(mod_reference: str) -> dict:
    try:
        (mod_key, version_key, *_,) = mod_reference.split(':')
    except ValueError as exc:
        (mod_key, version_key) = (mod_reference, 'latest')
    if version_key == 'latest':
        version_numbers = list(FULL_DISTRIBUTION[mod_key]['versions'].keys())
        version_key = version_numbers[0]
        for version in version_numbers[1:]:
            version_key = semver.max_ver(version_key, version)
    try:
        return (
            ('{mod_key}:{version_key}').format(
                mod_key=mod_key, version_key=version_key
            ),
            FULL_DISTRIBUTION[mod_key]['versions'][version_key]
        )
    except KeyError as exc:
        warnings.warn((
            'version `{version_key}` not found for mod `{mod_key}`, '
            'defaulting to latest'
        ).format(version_key=version_key, mod_key=mod_key))
        return get_specification(('{mod_key}:latest').format(mod_key=mod_key))
```

Listing 5.9: Mod Specification Retriever

Mod references in the context of the specified metadata use the notation `<mod-author>/<mod-name>:<mod-version>`. However, in some of the following examples, the `<mod-author>/<mod-name>` is substituted with a single uppercase letter. The process of building the dependency cone of a specified mod was automated using the above distribution structure along with the following function:

```
def build_install(
    mod_reference: str
) -> Tuple[Dict[str, List[Tuple[str]]], List[Tuple[str, str]]]:

    def list_dependencies(mod_reference: str) -> List[str]:
        dependencies = [get_specification(mod_reference)[0]]
        for dependency_clause in get_specification(mod_reference)[-1]['depends']:
            for dependency in dependency_clause:
                dependencies.extend(list_dependencies(dependency))
        return dependencies

    dependency_set = set(list_dependencies(mod_reference))
    conflict_pairs = []
    for dependency in dependency_set:
        for conflict in get_specification(dependency)[-1]['conflicts']:
            conflict = get_specification(conflict)[0]
            if conflict in dependency_set:
                conflict_pairs.append((dependency, conflict))
    return (mod_reference, {
        dependency: get_specification(dependency)[-1]['depends']
        for dependency in dependency_set
    }, conflict_pairs,)
```

Listing 5.10: Building Installation Information

From the returned `mod_reference`, `dependency_map`, and `conflict_pairs` triple, the values are given to the following function which builds a conjunctive normal form expression for the installability of `mod_reference` within the dependency cone of `mod_reference`. As shown in Definition 3.1.13, if a package p is installable with respect to the repository $(\Delta_R(p), D, C)$ then the package is installable with respect to the repository $R = (P, D, C)$. The optional `default` list allows the user to specify additional mods that are already installed and should **not** be uninstalled in order to make a valid solution.

```
def build_cnf(
    mod_reference: str,
    dependency_map: Dict[str, List[Tuple[str]]],
    conflict_pairs: List[Tuple[str, str]],
    default: List[str]=[]
) -> Tuple[Dict[str, int], Dict[int, str], List[List[int]]]:
    cnf_instance = []
    reference_map = {
        mod_reference: (idx + 1)
        for (idx, mod_reference) in enumerate(dependency_map.keys())
    }
    install_root = reference_map[get_specification(mod_reference)[0]]
    cnf_instance.append([install_root])
    for (mod_reference, dependency_list) in dependency_map.items():
        for dependency_set in dependency_list:
            dependency_clause = [
                -reference_map[mod_reference], *[
                    reference_map[get_specification(dependency_reference)[0]]
                    for dependency_reference in dependency_set
                ]
            ]
            if dependency_clause not in cnf_instance:
                cnf_instance.append(dependency_clause)
    for (conflict_reference1, conflict_reference2) in conflict_pairs:
        cnf_instance.append([
            -reference_map[conflict_reference1],
            -reference_map[conflict_reference2]
        ])
    for default_plugin in default:
        (mod_name, mod_specification,) = get_specification(default_plugin)
        if mod_name in reference_map:
            cnf_instance.append([reference_map[mod_name]])
    inverted_map = {v: k for (k, v) in reference_map.items()}
    return (reference_map, inverted_map, cnf_instance,)
```

Listing 5.11: Building CNF Formula

Because of how the premade SAT solver works, it is required to reference the mod and its dependencies as a list of positive integers. The returned `reference_map` and `inverted_map` are helpful for mapping between the mod reference and their assigned integers. The `cnf_instance` is the actual list of boolean clauses (represented as nested lists) that represents the satisfiability problem.

Utilizing a premade SAT solver, PicoSAT[2], it is possible to solve the generated `cnf_instance` using the following function:

```
def solve_install(
    mod_reference: str,
    dependency_map: Dict[str, List[Tuple[str]]],
    conflict_pairs: List[Tuple[str, str]],
    default: List[str]=[]
) -> List[Dict[str, bool]]:
    (reference_map, inverted_map, cnf_instance) = \
        build_cnf(
            mod_reference, dependency_map, conflict_pairs,
            default=default
        )
    inverted_map = {v: k for (k, v) in reference_map.items()}
    solutions = []
    for solution in pycosat.itersolve(cnf_instance):
        solutions.append({
            inverted_map[abs(idx)]: (idx > 0)
            for idx in solution
        })
    return solutions
```

Listing 5.12: Call SAT Solver

This function calls `build_cnf` itself, so a quick one-liner to solve an installation problem may look like the following:

```
solution = list(solve_install(*build_install('my-author/my-mod:latest')))
```

Proper load order was determined by using the following function when passed the output of the `build_install` function:

[2] `http://fmv.jku.at/picosat/`

```
def build_load_order(
    mod_reference: str,
    dependency_map: Dict[str, List[Tuple[str]]],
    conflict_pairs: List[Tuple[str, str]]
) -> List[str]:
    load_order = [mod_reference]

    def walk_dependencies(mod_reference: str):
        for dependency_clause in dependency_map[mod_reference]:
            for dependency in dependency_clause:
                dependency = get_specification(dependency)[0]
                if dependency in load_order:
                    load_order.remove(dependency)
                load_order.append(dependency)
                walk_dependencies(dependency)
        return load_order

    return list(reversed(walk_dependencies(mod_reference)))
```

Listing 5.13: Load Ordering

A simple `stdout` display function was also added in order to give the user an easier time understanding the output of the solution.

```
def display_solutions(mod_reference: str, default: List[str]=[]) -> None:
    reference = get_specification(mod_reference)[0]
    install = build_install(get_specification(mod_reference)[0])
    load_order = build_load_order(*install)
    solutions = list(solve_install(*install, default=default))
    if len(solutions) > 0:
        max_reference = max(len(reference) for reference in install[1].keys())
        for solution in solutions:
            print(('{0} \033[0;1mSOLUTION\033[0;0m {0}').format('-' * 20))
            mod_index = 0
            for mod in load_order:
                if mod in solution.keys() and solution[mod]:
                    print((
                        '\033[1;34m{0:#x}\033[0;0m: {1}:\033[1;34m{2}\033[0;0m'
                    ).format(mod_index, *mod.split(':')))
                    mod_index += 1
```

Listing 5.14: Solution Display

This function, when given a reference to a mod, builds, solves, and displays the results of the boolean satisfiability problem proposed by the `FULL_DISTRIBUTION` metadata.

Chapter 6 - Results

Now that the methods of solving dependencies within the context of Bethesda based mods has been laid out, it is possible to perform testing of the solver with both synthetic and real world examples. The following sections outline the results produced by several valuable examples of how dependency solving can benefit mod installation.

6.1 Synthetic Examples

The following are synthetic examples of the usefulness of the dependency solver discussed in 5.3. The simple dependency graphs are inspired from actual dependency graphs formed by the relationships between real mods. However, these have been simplified and adapted to better show the effectiveness of the dependency solver.

For an example of what the output of the following solutions look like, observe the below output for the trivial solution of the installation problem for `bethesda/update:latest`:

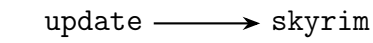


Figure 6.1: Trivial Dependency Mod Graph

The dependency cone formed by this mod is just a conjunctive dependency on `skyrim`. The solutions to this installation problem is shown below:

```
skyrim, update
```

The load order solution shown above is a comma separated list sorted by the mods which should be loaded first in order to avoid conflicts (load order). As shown in figure 6.1, because `skyrim` is listed as a master of `update` ($\texttt{update} \underset{1}{\rightarrow} \texttt{skyrim}$), `skyrim` must be loaded before `update` in order to avoid conflicts.

With the aid of the dependency solver functions defined in section 5.3, it is possible to show how mod management can utilize the practices behind package management. The following sections describe examples of mod relationships where this SAT solver may succeed and where it may fail. For the purpose of readability, *actual* mod names will be substituted with capital letters of the alphabet.

6.1.1 Successful Solving

Within this section, success is defined as properly proposing an installation solution for a given mod that both installs without file or asset conflicts and does not cause the game to crash. Again, because this solver is only accounting for explicitly declared dependencies and both file and asset conflicts, we do not expect every solution to be perfect. Rather, we expect the proposed SAT problem to be solved and present a valid mod installation solution.

Take for example Figure 6.2 which only contains dependencies.

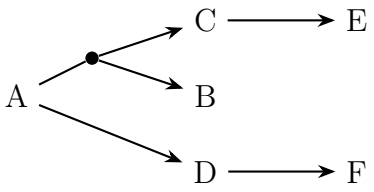


Figure 6.2: Simple Solvable Mod Dependency Graph

This graph is trivial to solve since in this case it is possible to simply install all dependencies alongside each other without conflicts occurring. Asking for the installation solutions of mod A using the `display_solutions` function returns 4 separate solutions.

Table 6.1: Load Order Solutions for Figure 6.2

`F, D, E, B, A`
`F, D, B, A`
`F, D, E, C, B, A`
`F, D, E, C, A`

As shown in the Table 6.1, the solver has found every installation set that is possible with the given dependency graph in Figure 6.2. Adding a conflict between two packages will cause the number of resulting installation sets to decrease. For example, observe the similar but slightly modified graph below:

Because there is now a conflict between B and D, no installation set where B and D coexist should be possible. Also, because the dependency from A to D is conjunctive, the installation of B should not be possible (due to the previous stated conflict). The following solutions is the result of giving this tree to the installation solver:

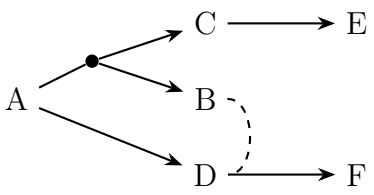


Figure 6.3: Simple Solvable Mod Dependency Graph with Conflicts

```
F, D, E, C, A
```

The results returned from the solver show that the only valid installation set in this dependency tree is the solution where B is simply not installed. Another example is shown where the conflict between B and D is removed and instead a conflict between C and F is defined.

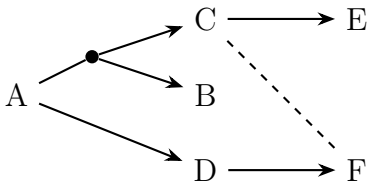


Figure 6.4: Simple Solvable Mod Dependency Graph with Conflicts

Because C now conflicts with a conjunctive dependency of A (F), the installation of C should never occur in a valid solution. The given solutions to this dependency tree are the following:

Table 6.3: Load Order Solutions for Figure 6.4

`F, D, E, B, A`
`F, D, B, A`

As shown in the solutions, two installation sets exist; one where only the dependency C is not installed, and one where the dependency along with its child (E) is not installed.

6.1.2 Solving Failures

For an example where the dependency solver should not find any valid solutions, consider the following dependency graph:

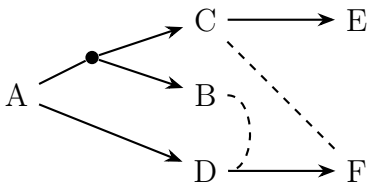


Figure 6.5: Simple Unsolvable Mod Dependency Graph

It is clear to see that this simple tree is not solvable. Because C conflicts with F and F is the conjunctive dependency of A, it is impossible to solve for A via the C subtree. And because B also conflicts with A's conjunctive dependency on D, it is impossible to solve via the B subtree. Therefore, the graph is not solvable. This can be confirmed by calling the `display_solutions` method and seeing that it produces no solutions.

6.2 Real World Application

It has been shown that the solver works on simple synthetic examples of installation problems within the `MOD_DISTRIBUTION` subset of mods. This section will display some tests

using the installation solver shown in section 5.3 on the dependency cone produced by actual mods. The plugins installed by default are defined within the `VANILLA_DISTRIBUITON` and the `DLL_DISTRIBUTION` as shown in Figure 6.6.

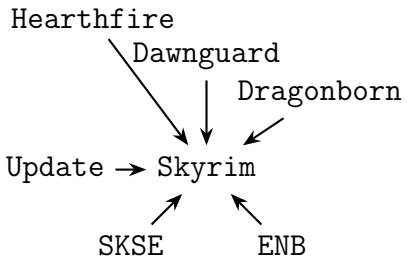


Figure 6.6: Default Distribution Dependency Graph

A good representation of a mod installation problem is represented by the graph shown in figure 6.7. Mod names have been generalized and dependencies can be either explicitly defined masters, or implicitly defined masters (via the author written documentation of the mod). The following graph describes the dependency and conflict relationships of the mod `npc-improvements` (most of the dependency relationships to `Skyrim` have been omitted as all mods depend on `Skyrim`).

After encoding these mod and their relationships into metadata entries within the `MOD_DISTRIBUTION` structure, the solution to installing `npc-improvements` produces the solutions found in table 6.5:

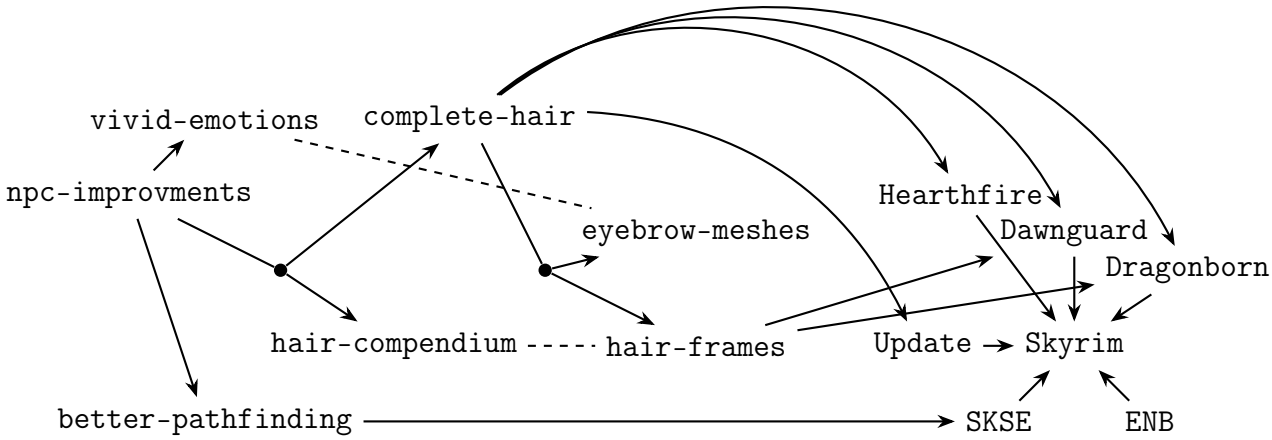


Figure 6.7: NPC Improvement Dependency Graph

Table 6.5: Load Order Solutions for Figure 6.7

1. skyrim 2. dawnguard 3. dragonborn 4. hair-frames 5. hearthfire 6. update 7. complete-hair 8. skse 9. better-pathfinding 10. vivid-emotions 11. npc-improvements	1. skyrim 2. dawnguard 3. dragonborn 4. hearthfire 5. update 6. hair-compendium 7. skse 8. better-pathfinding 9. vivid-emotions 10. npc-improvements

The load order solutions in Table 6.5 are shown slightly different to the solutions in previous tables due to longer mod names. Each column in the table lists an installation solution sorted by the required load order.

Note that these solutions are obtained by also providing the list of default mods as shown in Figure 6.6. However, the SAT expression is only taking into consideration Δ_R(`npc-improvements`). For this reason, `ENB` is never shown as installed in any solution. It is easy to see in Figure 6.7 that since `vivid-emotions` is a direct conjunctive dependency of `npc-improvements`, and that `eyebrow-meshes` conflicts with `vivid-emotions`, `eyebrow-meshes` can never be installed in a valid installation of `npc-improvements`. From the solutions returned from the solver, it is obvious that the two solutions come mainly from the disjunctive direct dependency from `npc-improvements` to `complete-hair` or `hair-compendium`. Both of these solutions are valid and function correctly after installation alongside the game.

6.3 Related Issues

The issues with current Bethesda-based mod distributions and some suggested solutions are detailed in the following sections.

6.3.1 Beneficial Metadata

Many of the issues surrounding mod management could be solved using proper author specified metadata. This mainly includes using some of the default metadata fields utilized in Debian package management (as shown in section 2.5.1). The following is a list of suggested metadata fields that if specified by the mod author, would make dependency solving and efficient mod distribution much easier. The fields with bold names should be required while others are optional.

- **Mod Name**: The name of the mod
- **Mod Author**: The author of the mod (mods are uniquely referenced by `<mod-author>/<mod-name>`)
- **Mod Version**: The structure version number of the mod (see 6.3.2)
- **Dependencies**: A list of dependency clauses (conjunctive and disjunctive which start the mod's dependency cone)
- Conflicts: A list of known conflicts
- Provides: A list of provided resources (similar to virtual packages)
- Suggests: A list of other mods that work well with the mod

Mods need to explicitly define their implicit dependencies at the very least for proper dependency solving. This is a simple but incredibly beneficial change that needs to be made for proper mod management practices. Current mod distributions have very little useful mod information tied to the mod itself. Because of this, most of the tasks for dependency solving must be done by hand. By implementing and enforcing basic metadata, most of the valuable features that are a part of traditional package management can also be designed and included in mod management.

6.3.2 Comparable Versioning

Another large problem that should be addressed when building a better mod distribution, is determining correct versions of mods. Current mod distributions do not have a well defined or enforced standard of version numbering and no accurate way of obtaining specific versions of mods. Therefore, if at any point a mod requires a specific version of another mod, there is no good way of correctly determining if the other mod is the required version. By simply enforcing each release of a mod to specify a new (greater) version number by utilizing either semver[1] or something similar, specific versions of mods could be accurately referenced by the combination of the mod's name along with a specified version number.

[1] `http://semver.org`

6.3.3 Automated Conflict Detection

Another incredibly beneficial but fairly complex feature would be the ability for a distribution of mods to routinely check for conflicts and populate metadata based on the results. The complexity of this comes from the various levels of conflicts that exist as mentioned in section 4.2.1. If a conflict is between two mods is automatically discovered, that information could be uesd to indicate what types of mods to prioritize checking for conflicts.

Chapter 7 - Conclusion and Future Work

In conclusion, the realm of package management will continue to evolve and sprout new complexities as more and more systems find it necessary or beneficial for the distribution and installation of user added functionality to be automated. The ability to easily install user written extensions to a particular environment encourages continual development and usage of that environment which benefits the entire community. Whether these systems require the distribution of simple or complex levels of components, most likely each and every system will need to build a solution for their specific environment. The main issues of package management still remain to be efficient dependency solving and proper package distribution. With the addition of more optimized solvers or better constructed distribution systems, the practice of automating the installation of component-based software will continue to grow.

The application of the practices discussed in this thesis regarding mod management is needed as more game publishers allow their games to be extended by their players. With the release of Fallout 4 and the re-release of Skyrim, Bethesda has provided an environment for modding its games not only on PCs but also on consoles. Because mods have now become a bigger part of a company's profits, mod management and the software described in this thesis needs to be standardized so that the practice of modding does not become as disorganized and unstructured as current mod distributions.

Bibliography

[1] "EDOS Project description," http://www.mancoosi.org/edos/, 2016, accessed: 2016-10-11.

[2] "Mancoosi Project description," http://www.mancoosi.org/, 2016, accessed: 2016-10-11.

[3] J. Boender, "A Formal Study of Free Software distributions," Ph.D. dissertation, Université Paris-Diderot - Paris VII, 2011. [Online]. Available: https://hal.archives-ouvertes.fr/tel-00698622/

[4] F. Mancinelli, J. Boender, R. Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the Complexity of Large Free and Open Source Package-Based Software Distributions," in *21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06)*. IEEE, 2006, pp. 199–208. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4019575

[5] P. Abate, R. Di Cosmo, C. Org Inria, R. Treinen, and S. Zacchiroli, "MPM: A Modular Package Manager," in *CBSE '11 Proceedings of the 14th international ACM Sigsoft symposium on Component based software engineering*. ACM, 2011, pp. 179–188. [Online]. Available: https://dl.acm.org/citation.cfm?id=2000255

[6] P. Trezentos, R. Dicosmo, S. Ere, M. Morgado, J. Abecasis, F. Mancinelli, and A. Oliveira, "New Generation of Linux Meta-installers," 2008. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.175.2105&rep=rep1&type=pdf

[7] R. Di Cosmo, "EDOS deliverable WP2-D2.1: Report on Formal Management of Software Dependencies," EDOS Project, Tech. Rep., 2008. [Online]. Available: https://hal.inria.fr/hal-00697463

[8] D. Spinellis, "Package Management Systems," *IEEE Software*, vol. 29, no. 2, pp. 84–86, 3 2012. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6155145

[9] P. Trezentos, I. Lynce, and A. L. Oliveira, "Apt-pbo: Solving the Software Dependency Problem using Pseudo-Boolean Optimization," in *IEEE/ACM International Conference on Automated Software Engineering (ASE'10)*, 2010, pp. 427–436. [Online]. Available: http://dl.acm.org/citation.cfm?id=1859087

[10] P. Abate, S. Cousin, O. Lhomme, C. Michel, J. C. Régin, and M. Rueher, "Deliverable D5.1 Description of the CUDF Format," Mancoosi Project, Tech. Rep., 2013. [Online]. Available: https://arxiv.org/pdf/0811.3621v1.pdf

[11] R. Di Cosmo, "EDOS deliverable WP2-D2.2: Report on Formal Management of Software Dependencies," EDOS Project, Tech. Rep., 2006. [Online]. Available: https://hal.inria.fr/hal-00697468

[12] M. Claes, T. Mens, and R. Di Cosmo, "A historical analysis of Debian package incompatibilities," in *IEEE International Working Conference on Mining Software Repositories*, vol. 2015-Augus, 2015, pp. 212–223. [Online]. Available: http://www.dicosmo.org/Articles/2015-MSR-coinstevol.pdf

[13] C. Szyperski, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addison-Wesley Professional, 2002. [Online]. Available: http://a.co/fE2Z2ls

[14] P. Abate, A. Guerreiro, Stéphane Laurière, R. Treinen, and S. Zacchiroli, "Manccosi Deliverable D5.2: Extension of an existing package manager to produce traces of ugradeability problems in CUDF format," 2010.

[15] P. Abate, J. Boender, R. Di Cosmo, and S. Zacchiroli, "Strong Dependencies between Software Components," in *Proceedings of 3rd International Symposium on Empirical Software Engineering and Measurement (ESEM'09)*, 2009. [Online]. Available: http://dl.acm.org/citation.cfm?id=1671258

[16] T. Lengauer and R. E. Tarjan, "A Fast Algorithm for Finding Dominators in a Flowgraph," *ACM Transactions on Programming Languages and Systems*, vol. 1, no. 1, pp. 121–141, 1979.

[17] A. Cicchetti, D. Di Ruscio, P. Pelliccione, A. Pierantonio, and S. Zacchiroli, "A Model Driven Approach to Upgrade Package-Based Software Systems," *CCIS*, vol. 69, pp. 262–276, 2010. [Online]. Available: http://www.mancoosi.org/papers/ccis10.pdf

[18] M. Vieira and D. Richardson, "Analyzing dependencies in large component-based systems," in *Proceedings 17th IEEE International Conference on Automated Software Engineering,*. IEEE Comput. Soc, 2002, pp. 241–244. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=1115020

[19] D. B. Tucker and S. Krishnamurthi, "Applying Module System Research to Package Management," *Tenth International Workshop on Software Configuration Management (SCM-10)*, p. 10, 2002. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.125.6917

[20] J. A. Stafford and A. L. Wolf, "Architecture-level dependence analysis in support of software maintenance," in *Proceedings of the third international workshop on Software architecture - ISAW '98*. New York, New York, USA: ACM Press, 1998, pp. 129–132. [Online]. Available: http://portal.acm.org/citation.cfm?doid=288408.288441

[21] J. Marques-Silva, J. Argelich, A. Graça, and I. Lynce, "Boolean Lexicographic Optimization," in *International RCRA Workshop*. Bologna: RCRA, 2010, pp. 1–15.

[22] ——, "Boolean lexicographic optimization: algorithms and applications," *Annals of Mathematics and Artificial Intelligence*, vol. 62, no. 3-4, pp. 317–343, 7 2011. [Online]. Available: http://link.springer.com/10.1007/s10472-011-9233-2

[23] J. Vouillon and R. Di Cosmo, "Broken Sets in Software Repository Evolution," *ICSE*, pp. 412–421, 2013. [Online]. Available: http://www.dicosmo.org/Articles/2013-DiCosmoVouillon-Icse.pdf

[24] P. Trezentos, "Comparison of PBO solvers in a dependency solving domain," vol. 29, pp. 23–3110, 2010.

[25] A. Barth, A. Di Carlo, R. Hertzog, L. Nussbaum, C. Schwarz, and I. Jackson, "Debian Developers Reference," 2016. [Online]. Available: https://www.debian.org/doc/manuals/developers-reference

[26] D. Le Berre and P. Rapicault, "Dependency management for the eclipse ecosystem," in *Proceedings of the 1st international workshop on Open component ecosystems - IWOCE '09*. New York, New York, USA: ACM Press, 2009, p. 21. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1595800.1595805

[27] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Dependency Solving: a Separate Concern in Component Evolution Management," 2012.

[28] J. Vouillon, M. Dogguy, and R. Di Cosmo, "Easing Software Component Repository Evolution," in *ICSE'14*. Hyderabad, India: ICSE, 2014. [Online]. Available: http://www.dicosmo.org/Articles/2014-DiCosmoMehdiVouillon-ICSE.pdf

[29] P. Abate, R. Di Cosmo, R. Treinen, and S. Zacchiroli, "Learning from the future of component repositories," *Science of Computer Programming*, vol. 90, no. PART B, pp. 93–115, 2014. [Online]. Available: http://www.dicosmo.org/Articles/2014-DiCosmoAbateTreinenZacchiroli-SCP.pdf

[30] J. Yu, W. Zhang, X. Tang, S. Li, Q. Li, and Q. Shen, "Measurement on a Peer-to-Peer Package Management System for Linux Distributions," 2014.

[31] P. Abate, R. Di Cosmo, L. Gesbert, F. Le Fessant, R. Treinen, and S. Zacchiroli, "Mining Component Repositories for Installability Issues," 2013.

[32] D. Le Berre and A. Parrain, "On SAT Technologies for dependency management and beyond," pp. 197–200, 2008. [Online]. Available: https://hal.archives-ouvertes.fr/hal-00870846/

[33] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "OPIUM: Optimal Package Install/Uninstall Manager," *29th International Conference on Software Engineering (ICSE'07)*, pp. 178–188, 2007. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4222580

[34] J. Mugler, T. Naughton, and S. Scott, "OSCAR Meta-Package System," in *19th International Symposium on High Performance Computing Systems and Applications (HPCS'05)*. IEEE, 2010, pp. 353–360. [Online]. Available: http://ieeexplore.ieee.org/document/1430094/

[35] J. Cappos, J. Samuel, S. Baker, and J. H. Hartman, "Package Management Security," 2009.

[36] F. Festi, "Package management system," 2010. [Online]. Available: https://www.google.com/patents/US8707293

[37] A. Athalye, R. Hristov, T. Nguyen, and Q. Nguyen, "Package Manager Security," 2014.

[38] R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in FOSS distributions," in *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades - HotSWUp '08*. New York, New York, USA: ACM Press, 2008, p. 1. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1490283.1490292

[39] V. Manquinho and J. Marques-silva, "PackUp : Tools for Package Upgradability Solving system description," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 8, pp. 89–94, 2012.

[40] Wikipedia, "Separation of Concerns," 2016. [Online]. Available: https://en.wikipedia.org/wiki/Separation_of_concerns

[41] J. A. Forbes, J. D. Stone, S. Parthasarathy, M. J. Toutonghi, and M. V. Sliger, "Software package management," 1998. [Online]. Available: https://www.google.com/patents/US6381742

[42] J. Argelich, D. L. Berre, I. Lynce, J. Marques-Silva, and P. Rapicault, "Solving Linux Upgradeability Problems Using Boolean Optimization," in *In Proceedings LoCoCo*, 7 2010, pp. 11–22. [Online]. Available: http://arxiv.org/abs/1007.1021

[43] A. Ignatiev, M. Janota, and J. Marques-Silva, "Towards efficient optimization in package management systems," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*. New York, New York, USA: ACM Press, 2014, pp. 745–755. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2568225.2568306

[44] R. Treinen and S. Zacchiroli, "Upgrade description formats: generalities and DUDF submission format," 2009. [Online]. Available: http://www.mancoosi.org/reports/tr1.pdf

[45] h. Shaull, "What's an example of an unsatisfiable 3-CNF formula?" 2014. [Online]. Available: http://cs.stackexchange.com/q/20118

Vita

Stephen Bunn was born to Eddie and Cyndy Bunn in 1995 in the state of North Carolina, United States of America. Homeschooled up until college, he was accepted by Appalachian State University in 2013 where he earned his Bachelor of Science in Computer Science in the spring of 2016. He then pursued a Master of Science in Computer Science using the 4+1 plan in order to graduate in 2017.