Chapter 1

Introduction

> **Man**: When you take apart a Lego house and mix the pieces into the bin, where does the house go?
> **Woman**: It's in the bin.
> **Man**: No, those are just pieces. They could become spaceships or trains. The house was just an arrangement. The arrangement doesn't stay with the pieces and it doesn't go anywhere else. It's just gone
> –XKCD Lego

When confronted with building a complex system, our first instinct in an attempt to understand and build the system is often to break it into smaller, simpler components. These components are defined by their goals in the overall system and their relationships to other components. Through this "divide and conquer" method, the design and construction of a complex system is broken down into a manageable problems to solve.

When building a complex system like a car, you would not attempt to build the car as a whole, but break it into parts for design and construction. A car requires a body, electrical system, interior, suspension and steering, engine .... The engine can be further broken into other parts like cooling, oil system, exhaust and intake systems, fuel, and so on. Each of these parts have relationships that lead to complex dependencies, for instance the carburetor blends the air from the intake with petrol from the fuel system.

The benefits of breaking a system apart are gained through the entire product process; design, implementation, deployment, maintenance and evolution.

The apocryphal story of George Washington's axe which has three times had its handle replaces and twice its head replaced deomstrates the power of maintence on a system of parts.

1.1 Software Components

There are more reasons for the use of components in software systems, additional benifits over their use in physical systems. Software components can be reused, replaced "on the fly" without interruption to the system, validated for correctness, and automatically building or modifying a system through resolving their defined dependencies.

In a physical car there are pumps which accomplish many different tasks, in a software system such replication is uneccesary so a single component can provide the servie to pump.

Replacing a tire or repairing a windshield while continueing to drive the car is impossible, but inside a software system it can be done.

If an enthusiast adds a turbo to a car, it may not work correctly in that context, it may create too much pressure and blow the engine. A component can have complex mechanisms like contracts or strictly defined specifications to automatically detect these problems.

Imagine being able to define the car you want by describing the functionality you require, and then a custom car, one that may exist no where else in the world, is automatically built for you. While then driving this car your requirements change, then the car changes parts to adapt to your new requirements. This is the power that dependency resolution offers component systems, automatic resolution of a users requirements.

1.2 Research Goals

The part of this vast research area we look at is the final point made in the previous section, the automatic composition of parts through resolving component dependencies and it's effects on a component system.

1.3 Overview of Thesis

Chapter 2

Background

2.1 What is a Component?

A component is best defined as

> A unit that can be composed This is a complete, but useless definition for our purposes, as Szyperski noted, a natural concept such as component is best defined by its goals not it's properties.
>
> Give examples of OSGi Bundles, Eclipse Plugins, DS Spring DM Debian Packages Maven POM CUDF
>
> Discuss differences and commonalities Multiple package installations Maven Debain No, OSGi Eclipse DS and CUDF yes Focus on interface not implementation dependence, Maven no, Debian Community defined virtual packages, OSGi kind of package name space related to installation, Eclipse Yes DS Yes. Version Control, All Yes Run time replacement Dynamic Dependence, Maven no (no runtime), Eclipse Kind of a bit buggy requires restarts, OSGI DS Debian Yes, CUDF is abstract so has no real implementation Conflicts, Maven?, Spring and DS no, OSGi Singleton, Debian and CUDF are Expressive Reccommendation, ??

2.2 Component Models

2.2.1 Debian

2.2.2 OSGi

2.3 Component Evolution

Versioning is the main mechanism in which components evolve. There is also branching, and environment specific altering.

Versioning components has significant research behind it.

Branching is when a difference in opinion or idealogy on how a component should proceed into the future causes the project to take two different routes given the same base code or idea.

Spring DM takes libraries and makes them OSGi compatable, Debian package can be compiled for different CPU architectures.

2.4 Component System Evolution

Version and Dynamic dependence, with focus on interface give extensive tools to the composer Add, remove, upgrade, downgrade components in the system while it is running.

There are some problems that must be solved within the component model like making sure that during the change, the system is never invalid or unstable. This is a scheduling problem as described by

The composer must also make sure that all dependencies are meet

2.5 Component Dependency Resolution

Chapter 3

Component Dependency Resolution

The strict definitions of the relationships between components can not only be used to find

3.1 Formal Definition

3.2 Optimisation

3.3 Current Implementations

3.3.1 Apache Maven

3.3.2 Nuget

3.3.3 RubyGems bundler

3.3.4 Paremus Nimble

3.3.5 OSGi Bundle Repostiory

3.3.6 Apt-get

3.3.7 Aptitude

3.4 Implementation with Boolean Satisfiability (SAT) Solvers

3.4.1 Davis-Putnam-Logemann-Loveland algorithm for SAT Solvers

3.4.2 CHAFF Watched Literals

3.4.3 Conflict Learning

3.4.4 Literal Order

3.5 Optimisation with SAT solvers

Chapter 4

Implementation

4.1 Common Upgradeability Description Format

4.2 Debian Model to CUDF mapping

The mapping of the Debian dpkg component meta-data to CUDF is mostly a direct process as the meta data is very similar. However, there are a few instances where the Debian semantic causes complication with the CUDF format, and in this section we describe these in detail. The first of these complications occuurs with versioning, as in Debian the version model is different to the CUDF model. The second instance is because of the special relationships that virtual packages have in the Debian model, these must be intergrated into the CUDF mapping. Lastly the priority ranking, recommends, suggests and other extra information that is not standard in the CUDF format, but with CUDF's extencibility can be represented and used in the resolution process.

The versions in Debian follow a model that first describes the epoch, the upstream version, then the Debian revision (further described in the Debian policy manual). When comparing two versions these values are compared lexicographically such that if the epoch is equal then the upstream versions are compared, and if these are equal then the Debian revision is compared. As the CUDF version model is merely a single integer number, the Debian versions are not easily mapped. To map these versions we order all Debian versions that are referenced in a repository into a single list, such that the least version has the index 0, and the last element in the list is the greatest

version. The index of the a Debian version in this list is then used as the CUDF version.

A virtual package in

4.3 OSGi Component Model

4.3.1 Mapping to CUDF

4.4 Comparison

Chapter 5

Investigations

5.1 Search Space Size

Chapter 6

Initial Investigations

6.1 Detailed comparison of OSGi and Debian

6.1.1 Graph analysis

6.2 Search Space Size

6.3 SAT solver optimisation

Chapter 7

Criteria

The two core criteria considered during component system evolution are the maximisation of the versions of individual components, and the minimisation of the change caused by the evolution to the system. Together these two criteria express that a system wants to be as up to date as possible, while changing as little as possible. Although these criteria must be considered in a CDR algorithm, they are not strictly defined, conflicting and for different users relate in different ways. In this chapter we explore the nature and definitions of the two criteria, and show that there is a significant gap between what is currently known and what we intend to explore.

7.1 Optimisation Notation

7.2 New Versions and Minimal Change

> Motion or change, and identity or rest, are the first and second secrets of nature: Motion and Rest. The whole code of her laws may be written on the thumbnail, or the signet of a ring.
>
> Ralph Waldo Emerson in "Nature", Essays, Second Series (1844)

Changing to what is newer and better, but being wary of the change because of cost and risk is a conflicting problem in many domains. For instance, in politics, conservitism is the philosophy that emphasises minimal and gradual change in society, where progressivism promotes change and reform to governments. These two conflicting ideaologies must be resolved, if a government is to function. As with component systems, the forces of change brought about by newer

versions of components and the resistance to change brought about by the harm it may cause are competing forces that must be resolved for a system to evolve. In this section we discuss the nature of these criteria, why they are important to consider and how they conflict.

7.2.1 Versions

A version is the mechanism through which individual components evolve. It is a unique marker which is compariable to other versions such that they are ordered.

The syntax and semantics of a version is usually defined within the component model. This forces all components that are developed for this model to follow the same general guidelines.

Although component must be versioned to conform to a versioning model defined by the component model, the way in which a developer uses this model to version their component is not usually strictly defined. Each component is designed to be an independent unit, as such developers of components are separate from one another, and the way in which they version their components can be different. Therefore, version comparison is only useful between different versions of the same component; e.g. a spell-checker component can be version 10 but a separate and superior spell-checker component may be only version 1. This makes using component versions to decide between components impossible.

As with the evolution of entire software systems, evolution of components requires constant maintenance. Bugs can be found, features can be added, and code can be refactored all which make the component better.

A newer version exists because the component has changed to be better in some way. This may be through a bug being fixed, a security hole being patched, or the functionality being extended. Having a newer version of a component in the system, may also make all of the other components that depend on it better as well, causing a propagation of system improvement with the upgrading of one component.

In Belief Revision, one must maintain the newest set of information while only changing the minimal amount of previous knowledge.

The change to the system that is caused when a newer version is installed may cause harm to your system. In the same way that a version upgrade can propagate benefits through the system, friction or errors can propagate and cause a faulty or not functional system.

7.2.2 Minimal Change

They are also not strictly defined as the mechanisms to either compare two systems versions or the change from a system, can be measured in different ways.

The measure to use when determining version of a system from the versions of installed components varies. As a set of components can be versioned differently, we need to explore functions that can aggregate the versions of different components together in a meaningful way.

The Mancoosi organisation uses a metric that minimises the number of components in a system that have better versions, e.g. a system $a-1, b-1$ if $a-2$ exists is 1 out of date. The Eclipse P2 implementation counts the amount of versions that are better for each component, e.g. a system $a-1, b-1$ if $a-2$ and $a-3$ exist, is 2 out of date. Both of these take into account only the component and not the components that depend on it.

The measurement of change that a system goes through during evolution is also difficult to define. The added, removed, updated, total changed, have all been considered as metrics before. As with the version metrics, none of these consider the dependencies when looking at the change that is performed on the system.

7.2.3 Versions, Minimality and Users

7.3 The Gap

Each time a user decides to evolve a component system, the decision must be made about the risks of the evolution. In an environment which is mission critical, all risk is eliminated and an unecessary change to the system is too risky. In a development environment where the user may be trying to fix potential problem, or test different packages, then the risk is accepted as the system is essentially disposable, and a complete re-installation is not out of the question. These two strategies are represented in by the current strategies used, however very little middle ground is avaiable for the customisation of applications. The user may want

Chapter 8

Simulation

To evaluate different strategies to evolve a component system, we simulate the environment in which this evolution occurs. This simulation models the user-interaction, the repository, and the system to draw conclusions on the benefits and draw backs of a particular evolution strategy. To ensure that the model is credible we use a methodology outlined by, The data that is used was extracted from logs of real users, a user survey conducted on a popular Internet forum, component and temporal information collected from repositories, meta-information about those repositories.

In this simulation we take a component system and evolve it over time using real and approximate information and look at the resulting systems.

In this chapter we first describe our motivation for using a simulation, then we present our methodology first describing our formulation of the problem, our reasons and methods for the collection of data, discuss our simulation model validity when compared to the real world. The design of the experiments are then described and the variables assigned values, finally we present the results from this simulation.

8.1 Why Simulate?

To evaluate an evolution strategy we could either look at a set of actual systems with real users and collect data, or we could simulate the necessary aspects as realistically as possible, then study the results. As using real users with real systems will always return more valid results to simulating, why would we opt to simulate?

The main reason for simulating is that finding enough users who would allow an experimental component resolution algorithm to alter their real system would be extremely difficult. A user will likely not trust a newly released resolver, as their system is important to them and even the slightest error can cause a system to become unstable. To gain the trust of possible users the experimental resolvers would have to be thoroughly tested through a repositories development cycle and be well maintained. Moving a package through this cycle can be itself a massive undertaking lasting several months. For Eclipse Plug-ins it involves going a component going through four phases Proposal, Incubation, Mature, and Top-Level [1]; similar to Debian's process of moving through the phases Unstable, Testing, Frozen, and finally stable [2]. After it has been through this cycle maintenance of the resolver is still required; for instance the resolver apt-get since its initial release has released more than 2 versions a month [3]. This shows how much effort must be applied to earn a users trust, and this effort may outway the actual benefits when simulation is a cheaper alternative.

We could use a case study of users where users evolve systems in a monitored environment. This is also probalematic because

Simulation is a surrogate of the real system, such that it represents the aspects core to the problem, the evolution of component systems. As it is only an approximation of the real world, the accuracy to which it actually represents the real world is not 100%. The goal is then to make it a "close enough" approximation so that the conclusions drawn from it are valid in the real world. So when analysing the results and forming conclusions, we must take into account the assumptions we have made creating the simulation to determine the conclusions overall accuracy. Therefore, the majority of the effort when creating a simulation is gathering and using valid information when making assumptions.

8.2 Methodology

Creating a simulation that accurately represents the evolution of a component system so that we can analyse the effect different evolution strategies have on a systems properties is the core objective of this simulation. To this end, we follow the a methodology outlined by, This gives us a set of guidelines to follow when preparing a valid simulation including; specifically defining the problem, building a conceptual model, collecting data,

[1] http://www.eclipse.org/projects/dev_process/development_process_2010.php
[2] http://en.wikipedia.org/wiki/File:Debian-package-cycl.svg
[3] http://changelogs.ubuntu.com/

8.3 Conceptual Model

To create our conceptual model, we must first precisely describe the problem; When evolving a component-based system, different strategies can be employed by the user and the resolver. These strategies have unknown effects over a long period in which the user evolves the system. Through simulating and analysing these effects we can choose a strategy more effectively. There are two parts to these strategies, first how the user interacts with a resolver, secondly how the resolver handles these requests.

8.3.1 Strategies

A user interacts with a resolver to evolve a system by installing, removing or updating packages. The way in which these interactions occur is usually related to the purpose of the system, and what the type of user. For instance, a server administrator is less likely to install a component into a system as the system has only one task and if it is working their is no need to change it, however a desktop user will more likely install different packages because they use their system for many different tasks. How a user updates their system is also different along these lines, server admins. will likely only update when absolutely necessary as if it is not broken why fix it. Yet desktop users will update more frequently as they want the best system they can have, and new features or better performance are usually wanted.

As described in previous chapters, dependency resolvers often try to minimise change and maximise the versions. These two objectives have been implemented in many different ways, how these interact with each other is the strategy a resolver employs. For instance, the trendy strategy from the Mancoosi organisation describes the strategy to minimise the removed components above all other criteria, this is a common strategy because removal of a component is seen as very risky.

8.3.2 Effects

8.3.3 Configuration

A configuration is the set of variables which define the parameters of the simulation. How these variables are defined are derived from what goals of the simulation. As the goal of this simulation is to look at different resolver and user strategies, these are the core changes that are made. Given that we want to look at the effects of many

different possible ways a system can evolve, we also change the packages that are selected to be installed by the user. The variables like, what repositories are used, and over what time period are seen as not being relevant to our eventual analysis, therefore we do not change them and leave them static

Given this simulation is looking at the effects of the strategies employed by users and resolvers on the component systems, we must look at different resolvers, and different user strategies. What packages the simulated user selects to install and

We are testing resolver strategies P2, trendy, Hamming + max version, PageRank,...

The user installs different amount of packages per day, and updates at different intervals.

We are selecting the packages to install that are different.

Time, We have decided not to

8.3.4 Sensitivity Analysis

Through altering some of the configuration variables a small amounts then running the simulation, the variables that are most important will alter the results significantly. This is an important step in our simulation as it shows us the variables that we must exert more effort to get correct.

8.4 Data-sets

The most complex and difficult part of this (or any) simulation is the human component, the user, as such that is where much of our data collection focused. Data has been collected from a user survey which was performed on a popular Internet forum, user logs from resolvers were collected with the survey. The Ubuntu popularity contest is used to determine package popularity, as well as a user forum thread where users posted their top ten packages. A package that contains a list of applications in the Ubuntu repository was included. The entire Ubuntu repository, with relevant date information about packages was used.

A user survey was conducted online via a popular Internet forum http://reddit.com/r/ubuntu, this involved nearly 60 participants. The survey focused on user interaction with a resolver and the lifecycle that is associated with their component system. The results are summarised below,

Resolvers often keep logs of their activites, these usually only include the changes to the systems that are made, and not what the usre requested. Therefore, some of the information that can be obtained

The Ubuntu Debian popularity contest[4] is an excellent, accurate and broad data-set of information of the popularity of Ubuntu packages. Each week this automated survey is submitted by nearly two million users including the currently installed packages they have on their system.

The package app-install-data contains a list of applications, that are available in the repository, which the user may wish to install. This is useful for applications to use that help users search for and find an application that they may wish to install, like the Ubuntu Software Center. This comprehensive list currently[5] contains 2393 applications.

The Ubuntu Repository, as with most open and free software, is freely downloadable. It contains all the packages that have ever been in the repository with the information of when the package was added. We used a web scraper to download all the packages, and then we extracted their control files (the meta information file) and converted it to CUDF as previsouly described in section. As the date a package was uploaded, we used the extensible CUDF syntax to include what the date when they were uploaded.

One of the aspects that is critical to a simulation is a time over which it is occuring, so the starting system is important aspect for this simulation. Ubuntu has 6 monthly releases one, in April and one in October, the syntax of the version of each release is first the year, then the month in which it was released, e.g. 10.04 is the release in April 2010. Given that we are running this simulation over the course of a year, we are selecting that year to be

In our abstraction of reality the user has two actions with the resolver, either to install a new package or to try upgrade the entire system. In reality the user can do much more fine grained actions, where they can remove packages, upgrade individual packages or even create complex queries for the resolver to solve. These additional actions, along with being difficult to simulate, are rarely performed by a user, as found with our user survey. How often a user installs a new package is found through looking at user logs, and how often a user updates a system is found through our user survey. These pieces of information allow us to approximately represent real users.

[4] http://popcon.ubuntu.com/
[5] May 24th 2011

8.5 User Approximation

Installation Distributions from the Logs, Actions and life cycle from the survey.

8.6 Package Popularity

Determining the probability a user will select a package to be installed is difficult given the enormous amount of factors this relies on. The users job, location, current tasks, previously installed software, favourite colour... all may effect when and what package a user selects to install. Given that all this information is impractical to simulate, we abstract this problem into the form of two questions; what packages may a user select to install and how likely would a user select to install these. We attempt to answer these questions through using the set of packages listed in the package app-install-package weighted with their popularity from the Ubuntu popularity contest. This method has some draw backs, not all packages a user may install are listed and there is no correlation between packages selected for install. However, by limiting our approach we have answered these questions without sacrificing much simulation integrity.

Many of the packages in a repository are not ones which a user would directly select to install. Most packages provide libraries, background daemons, interfaces between services; packages that are only needed through dependencies. A user would not likely install these as they do not directly allow the user to complete tasks in the system. The list of applications from the package app-install-data is used to answer the first question, what packages may a user select to install.

The second question is then answered through analysing the dataset and determining the probability a user will have a particular package installed. The reason why we cannot use this information to also answer the first question is that many of the most packages installed are there because they are depended on by many different applications, e.g. a media library that decodes a stream may be used by many different media playing applications therefore installed on many users systems, ranking it high in the popularity contest but it was never directly installed by the user.

This relies on the fact that a user would like to

Although a user will more likely install this list can be assumed to be a complete list of applications the main problem is that more experienced users may directly select to install packages that are not deemed applications. For instance, the package build-essential

includes tools in which a user can build their own Debian packages, this is a task for many users, though not deemed an application therefore not included.

The core problem with this probability, is that it doesnt measure corrolation between packages being installed. For instance, i

8.6.1 Failed Attempts at ranking

In creating this set packages weighted to their popularity, we also know the Ubuntu popularity contest is an excellent accurate and broad data-set of information with one main draw back of having superfluous packages, we attempted to use other means to eliminate these packages and then

To estimate the popularity of a package, the Google API for automated search completion is used. When given a query, this API returns an estimate of the amount this query has been searched for by other users, it also returns a list of related searches that users have searched for. What query is used is the most important aspect of this approach, searching merely for the name of the package may return user queries from other domains, e.g. searching for the package "wine" may return oenophile sites. Also using multiple different queries can allow for a more robust heuristic as it allows measuring their popularity from different perspectives, as one query may not be used when users search for a particular package.

This is a very general approach, it involves many aspects that effect the results making them possibly inaccurate. Therefore, the estimates are validated against a popular Ubuntu forums thread[6] that asks the user to post the top packages they install in their ubunutu systems. The results of this thread are tallied to compare against the google approach.

We have selected three queries to build our heuristic: "apt-get install", "ubuntu" and "install",

8.7 Repository

The collection, mapping and use of the Ubuntu Repositories. Version mapping

[6] http://ubuntuforums.org/showthread.php?t=35208

Chapter 9

Conclusion

9.1 Related Work

9.2 Future Work

Chapter 10

appendix

10.1 Glossary

Composer: The user who creates or alters component systems (compositions)

Versioning Model: A method to define and compare versions