Chapter 1

Introduction

When confronted with a large complex system, the first instinct, in an attempt to understand it, is to break it into smaller, simpler units, then examine each part individually while trying to understand the relationships between them. Through doing this a complex system as a whole is able to be grasped through layers of abstraction. Looking at the human anatomy, we can look at the nervous system, the cardiovascular system, the muscles, the skeleton . . . , each individual part with a distinct function and interactions with other parts. Understanding a system as a set of units and interactions gives us tools for the study and maintenance of problems.

In complex software systems, understanding the entire system as a set of parts with relationships has many of the same benefits analogous to the anatomy example. It allows specialization, by splitting the system into parts experts in those parts can focus on their own domain and only have to know how it interacts with other parts.

This same method is used when creating complex systems, first we divide the parts into individual functional units, then define the relationships between the units.

Component Software consoldates code into units

1.1 Overview of Thesis

First Background, Then define what a component is based on literature and examples

Chapter 2

Background

2.1 What is a Component?

Discuss what a component is and who defines it as such

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 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.

2.3 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

Chapter 3

Component Dependency Resolution

Now we know the problems

3.1 Formal Definition

3.2 Optimisation Notation

3.3 CUDF

3.4 CUDF Mapping

Chapter 4

Implementation

4.1 SAT Solver

4.1.1 DPLL

4.1.2 CHAFF Watched Literals

4.1.3 Conflict Learning

4.1.4 Literal Order

4.2 Optimisation Representation

Chapter 5

Component Models

5.1 Ubuntu Component Model

5.1.1 Mapping to CUDF

5.2 OSGi Component Model

5.2.1 Mapping to CUDF

5.3 Comparison

Chapter 6

Investigations

6.1 Search Space Size

6.2 SAT Optimisation

6.2.1 Literal Order

6.2.2 Learning Choices

Chapter 7

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, and from a Google API.

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

7.1 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 created more than 2 versions a month [3]. The users trust is hard earned and this effort to use real users may outway the actual benefits when simulation is a cheaper alternative.

Simulation is a surrogate of the real system, such that it representes only the aspects that are 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 analyising the results and forming conclusions, we must take into account the simulations accuracy. So the effort of creating a simulation has two aspects, the experimentation and the validation of the simulation.

7.2 Objectives

7.3 Design and Implementation

For this

7.3.1 Drawbacks if this Simulation

The installation of different packages are not independant of each other.

The selection of popular packages has a few drawbacks.

[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/

7.4 Data Collection

7.4.1 Survey

7.4.2 Log Analysis

7.4.3 Repository Collection

7.4.4 Validation

7.5 User Survey

7.6 Log analysis

7.7 Criteria

7.8 Heuristics

7.9 Experiment

7.10 Results

7.11 Analysis

7.12 Experiment Conclusions

Chapter 8

Conclusion

8.1 Related Work

8.2 Future Work

8.3 Glossary

Composer: The user who creates or alters component systems (compositions)
Versioning Model: A method to define and compare versions