An Empirical Study into Component System Evolution

Graham Jenson, Jens Dietrich, Hans W. Guesgen, Stephen Marsland

School of Engineering and Advanced Technology
Massey University
New Zealand
GrahamJenson@maori.geek.nz, J.B.Dietrich@massey.ac.nz,
H.W.Guesgen@massey.ac.nz, S.R.Marsland@massey.ac.nz

Abstract. Changing only what is necessary during system evolution is an attractive feature of component-based systems. This minimal system change during component system evolution is left un-standardised, yet it is clear that the definition of minimal will have significant effects during evolution. The aim of this paper is to empirically test different definitions of minimality, to explore and compare their properties and find an appropriate definition.
Through defining different types of minimal change, we use the Ubuntu GNU/Linux distribution to simulate component system evolution by selecting components to install and calculating the resulting system. We then extract the resulting system properties to compare our different minimal types. The results of this experiment not only show some invariant properties of component system evolution, but also that simple definitions of minimality can be as good as other more complex methods.
Key words: Dependency Resolution, Component Distribution, Evolution, Minimal Change

1 Introduction

Evolving component-based software requires only altering a minimal set of parts of the system, where as evolving a monolithic piece of software is achieved by replacing the entire system. This comparison is described by Szyperski [1] as "evolution not revolution". As introducing change to a system introduces significant risk, minimising change during evolution lowers the risk of component-based systems compared to monolithic systems. This risk occurs due to side effects that are caused by change, and changing large parts of a system incurs more side effects versus altering only specific parts of a system. Therefore, lessening the change lowers the risk, and this is a very attractive feature of using and developing component-based software.

However, calculating a minimal change during evolution can be difficult and complex, as any desired change may be potentially satisfied by many different solutions [2], making finding a minimal system a combinatorial optimisation problem. Also exactly what is meant by "minimal" is often not specified, yet clearly how it is defined will have a significant effect on how the system evolves. Therefore, finding a definition of “minimal” that is suitable for use by component systems requires looking at its effects on the systems throughout their evolution. The goal of this paper is to look at different types of minimal change, and simulate evolving a component system to find the consequences of their use, and to identify an appropriate minimal definition.

This process of component system evolution can be automated into the function Component Dependency Resolution (CDR) [2]. This takes advantage of the explicitly declared requirements of components to find a system where all component constraints are satisfied. CDR can be used at design time to determine the required dependencies to build and test a project (as in Apache Maven [3]), at run time to evolve or extend a component-based system (as in Eclipse P2 [4]), or it can be used to build and restructure software product lines [5]. CDR is also used in package management systems, like apt-get [6], for adding, removing and updating components in package based operating systems like Ubuntu[1].

2 Minimal Change Definitions

2.1 Formalising CDR

First we denote the set of all components as C, then define CDR as the function $C_R : 2^C \to 2^C$ such that it takes a set of user-requested components (a set of user requests is denoted as X, and an individual request is denoted Δ), and returns a set of components that satisfy all the component constraints R. For instance, given $X = a$, $C_R(X)$ would try to resolve a set of component that contains a while considering the component constraints R that state if component a is in the set then b or c must also be in the set. Components are usually described with a name and a version, such that no two components can have the same name and version. The user requested components must be in the system i.e. $X \subseteq C_R(X)$ as the user requirements are absolute. The returned set of components is seen as the resulting system of installed components, e.g. if $a \in C_R(X)$ then a is installed.

The constraint types in R are described as either:

- dependency requirements in the form $a \to c_1 \vee \ldots \vee c_n$ e.g. $a \to b \vee c$ means a depends on b or c
- conflict constraints in the form $a \to \neg c$, meaning a conflicts with c

These constraints are defined semantically such that: a dependency constraint $a \to c_1 \vee \ldots \vee c_n$ means if $a \in C_R(X)$ then $c_1 \in C_R(X)$ or $\ldots$ or $c_n \in C_R(X)$, and a conflict constraint $a \to \neg c$ means if $a \in C_R(X)$ then $c \notin C_R(X)$. These types of constraints are expressive enough to describe most CDR problems.

Expressing CDR w.r.t. minimality is done using a distance function denoted as $d(C', C'')$, which takes two sets of components and returns some measure of the distance between them. A system C' is said to be minimal w.r.t. to the

[1] http://www.ubuntu.com

original system $C_R(X)$ if there does not exist another system C'' such that $d(C_R(X), C'') < d(C_R(X), C')$. The function $d(C', C')$ will always return a minimal element w.r.t. the order $<$. We then use different definitions of d to define different types of minimal criteria.

2.2 Minimal Definitions

Here we define four different minimal types:

1. Hamming, based on the Hamming distance
2. Paranoid, optimisation function defined for the Mancoosi project
3. P2, used in the Eclipse IDE platform
4. PageRank, defined with the PageRank with priors algorithm

The **Hamming** distance [7] is used here to define minimality of component systems. This distance is defined as the size of the symmetric difference of two sets C_1, C_2, i.e. $d_H(C_1, C_2) = |C_1 \ominus C_2|$. This is a greedy search for minimality, as selecting the minimal system based only on the component changes from the current system may result in more changes over the long term.

The **Paranoid** distance function, defined by the Mancoosi (*MANaging the COmplexity of the Open Source Infrastructure*) project[2], is the lexicographic composition of two distance functions *removed* and *changed*. It is defined such that each component has a name $C \to Name$, where the function $V : 2^C \times Name \to 2^C$ returns all components with a name in a set of components (these are the different versions of a component). They are defined such that $removed(C_1, C_2) = |\{name \mid V(C_1, name)$ is nonempty and $V(C_2, name)$ is empty $\}|$, and $changed(C_1, C_2) = |\{name \mid V(C_1, name)$ is different to $V(C_2, name)\}|$. The removed function therefore only considers when all components of a name are removed.

The distance function is then defined $d_P(C_1, C_2)$ to return a pair (a, b) such that $a = removed(C_1, C_2)$ and $b = changed(C_1, C_2)$. The distance order $\leq_P$ is lexicographically defined with respect to $C_R(X)$ as given two models C_1, C_2, (a, b) equals $d_P(C_R(X), C_1)$ and (a', b') equals $d_P(C_R(X), C_2)$ then $C_1 \leq_P C_2$ iff $a < a' \vee (a = a' \wedge b \leq b')$. This definition states that removing a component is infinitely worse than adding one, therefore we should expect this to generate larger solutions.

Eclipse **P2** [8, 4] is the provisioning system for the Eclipse IDE platform. Its optimisation function minimises the removed components, while maximising the version of installed components, we denote it's distance function as d_{P2}. This function is fully described by Le Berre and Parrin in [4]. The goal of combining these two criteria is to simultaneously update components while altering the system. This strategy can be seen as a preemptive change, so that a user will not need to later update currently installed components, therefore minimising future change. As this is the only minimal criteria that includes the goal to maximise the currently installed version we expect more change during evolution.

[2] http://www.mancoosi.org/

A components dependence on another component creates a tangible connection that should be considered when evolving a system. This connection can be seen as a form of trust, given a developer selects the components to have their components depend on, or as a measure of importance to a system, as a component that is highly depended on has more responsibility to the system. This should be considered when making the choice to either remove or add a component to a system. By analysing dependencies between components we can give a weight to a component-based on the structure of its relationships.

The structure of component dependencies can be abstracted to a graph, and by analysing the importance of nodes in this graph using the **PageRank** with priors function [9], we attempt to quantify the value of a component in a system. PageRank with priors estimates the probability of stopping on a node in a random walk over a directed graph given a set of starting nodes. We see this as an approximation to the probability that a component is depended on in a system. A directed graph $G = (V, E)$ is generated from the dependency rules such that V is the set of components, and $(a, b) \in E$ iff there is a dependency constraint in R where $a \rightarrow c_1 \vee \ldots \vee c_n$ where $b = c_i$ for some i. The function $PR(a)$ denotes the PageRank of a, so given the set $C_R(X) \cup \{\Delta\}$ is the starting set of nodes, and the generated graph G, the distance function $d_{PR}(C_1, C_2) = \sum_{a \in (C_1 \ominus C_2)} PR(a)$, i.e. the distance is the sum of the PageRank values in the symmetric difference of the systems. This distance is the only one to consider the components relationships when calculating the effect of removing or adding them. Our hope is that by minimising the impact on the structure of the system, this will minimise the necessary changes over a long period of time.

2.3 Example

Here we give an example to demonstrate these minimality definitions; it forces a choice between either removing a single component b or adding two components e and f.

Given $X = \{a\}$, $C_R(X) = \{a, b\}$, $R = \{a \rightarrow b \vee c, c \rightarrow \neg b, d \rightarrow c \vee e, e \rightarrow f\}$, and $\Delta = d$. The main choice when fulfilling the user request to install d is between installing c and removing b, or installing both e and f. The solutions to this are $C_1 = \{a, b, d, e, f\}$, $C_2 = \{a, d, c\}$, $C_3 = \{a, d, f, c\}$ and $C_4 = \{a, d, e, f, c\}$. The minimal distances for each possible solution is given in table 1.

Solution (C_n)	C_1	C_2	C_3	C_4
$d_H(C_R(X), C_n)$:	3	3	4	5
$d_P(C_R(X), C_n)$:	(0,3)	(1,3)	(1,4)	(1,5)
$d_{P2}(C_R(X), C_n)$:	0	1	1	1
$d_{PR}(C_R(X), C_n)$:	0.343	0.657	0.709	0.785

Table 1. Example showing the different minimal distance results

Firstly C_1 and C_2 are both minimal given the Hamming minimal criteria. The paranoid criterion sees removing a component as infinitely worse than adding a component, and P2 only considers removed components, so C_1 is their minimal solution. C_1 is also the minimal solution for PageRank distance, this is because adding the component c is seen as structurally changing the system significantly.

We can alter the problem slightly by stating the c is the same component as b but a better version. This would only change the outcome of the Paranoid and P2 minimal types as they depend on the component versions. Paranoid would no longer see solution C_2 as removing b but replacing it with a better version. This reduces the distance of C_2 from $(1, 3)$ to $(0, 3)$, making solutions C_1 and C_2 both minimal. The P2 minimal solution would also decrease the distance to C_2, but enough to make C_1 no longer minimal, as replacing the lesser version b with c is preferred.

2.4 Implementations

To describe the CDR problem in a standard way we use the Common Upgradeability Description Format (CUDF) [10]. This specification was developed by the Mancoosi project as an abstract way of describing package upgradeability problems in open source GNU/Linux distributions. The Mancoosi project provides a range of tools with this specification, enabling us to create, modify and validate the generated solutions for consistency.

For the P2 and paranoid minimal definitions the implementation used is P2CUDF, from Le Berre and Parrin [8] of the Eclipse P2 provisioning system. This has been validated as an effective solver through the Mancoosi International Solver Competition[3].

The implementation of the minimal Hamming distance, and the minimal PageRank functions is based on the P2CUDF implementation. Like P2CUDF, it uses the SAT4J[4] Boolean satisfiability solver, with Pseudo Boolean Optimisation (PBO) described by Le Berre and Parrin [8]. This method converts a CDR problem expressed in the CUDF format into a Boolean satisfiability problem, iterating over finding a solution and adding a constraint that ensures the next solution found will be more optimal, until no more solutions can be found, then returns the last solution identified. To calculate the PageRank with priors values we use the efficient implementation from the JUNG[5] library.

3 Experiment

The core of this experiment is to iteratively evolve a component system using randomly selected user requests. Through evolving many systems over many generations, we are able to identify properties of this evolution.

[3] http://www.mancoosi.org/misc/
[4] http://www.sat4j.org
[5] http://jung.sourceforge.net/

For our experiments our initial system is the default installation of desktop Ubuntu 10.10 “Maverick Meerkat” operating system, this contains initially 1,282 packages. The total set of packages we use is the set gained from the initial Ubuntu repositories (main, updates and security) updated on 16th of December 2010; this includes 33,113 packages. This set of components is used because they form a large and complex system that has many users, and other such systems may not have the amount of components or complex interactions to extract useful information from. However, the ability to generalize the results of this experiment to other component models may be possible, as many component models may have similar component relationship structure.

In this experiment we create 100 evolutions each with 150 user requests denoted as $e_i = \{\Delta_1^i, \ldots, \Delta_n^i\}$ where $n = 150$, for $i = 1$ to 100. A generation of an evolution e_i is denoted as g^i, is then defined such that $g_j^i = C_R((X_{j-1}) \cup \Delta_j^i)$ w.r.t. $C_R(X_{j-1})$. The initial system, denoted as g_0, is the starting components of Ubuntu 10.10, and g_1^i to g_n^i are generations created by requests Δ_1^i to Δ_n^i. The same set of user requests are given to all definitions of minimality in the same order. Given that our four different definitions of minimal, 100 evolutions each with 150 user requests, we solve a total of 60,000 CDR problems for this experiment.

Each of the user requests is randomly selected from the set of 10,000 most popular Debian packages, restricted to 8,492 after packages not contained in the default repositories are removed. The measure of popularity is from the Debian popularity contest[6] sorted by vote, this was collected on December 16th 2010. This data is only the name of the package, not the version, so if multiple versions are available of a package we select the highest version as the change.

The main experimental challenge that should be discussed is that although two or more models can be minimal given any definition, the implementation of CDR can only return one system. This may effect some of the results, however, we hope it is mitigated by the large amount of random generations and problems used.

4 Results and Analysis

The presentation of our results is focused on the change of components, i.e. the symmetric distance, between generations.

The first aspect that we will discuss is the failure rate, caused when a set of user requests is unsatisfiable (this is minimal order independent). Out of the 100 evolutionary problems 29 failed, these failures were removed from results discussed here as they obfuscate other details. The trend of these failures increases through generations, where over 50% of the failures occurred in the last 50 user requests. This is because the systems become more constrained as more user requests must be satisfied, meaning that there is a greater chance of conflicting constraints. The reasons for these failures has not been explored, it may be that there are common reasons that could be eliminated.

[6] http://popcon.debian.org/

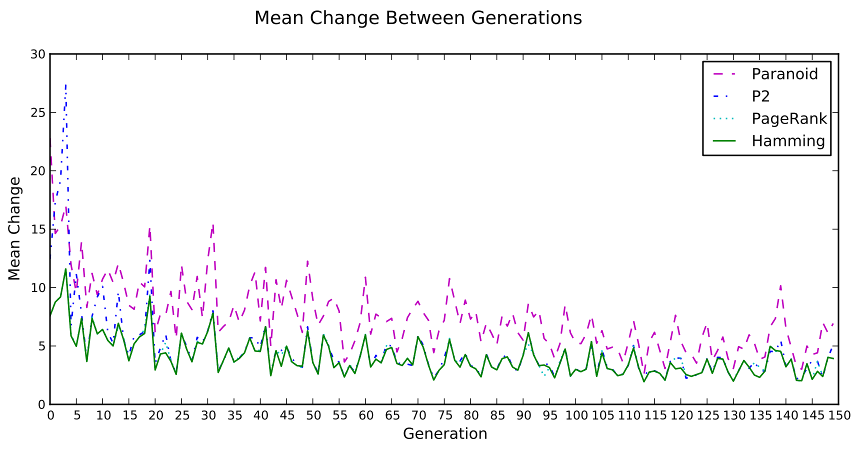

Fig. 1. The comparison of the minimal definitions mean change per generation. This clearly shows that the P2 solver initially changes the system significantly, then mimics the Hamming distance. PageRank is very similar to the Hamming minimality, and Paranoid constantly changes more than the other definitions.

Another invariant result is that the majority of user requests require very few changes to the system, however occasionally significant events occur that require large changes to the system. This can be seen in table 2 where the average change for all minimal types is less than 5, but they have wide standard deviations and high maximums values. These significant events tend to happen earlier in the systems evolution, this may be because there is more that can be changed in a less evolved system. This is seen by looking at the mean maximum changes between generations 1 to 30; which for Hamming is 71, Paranoid 102, P2 172, and PageRank 71; compared to the maximum mean change from generations 30 to 150; which for Hamming is 42, Paranoid 74, P2 46, and PageRank 44. These values show a clear decrease in large changes later in the system evolution.

Minimality	Mean Time (s)	Mean (SD)	Standard Deviation of (SD)	Maximum (SD)
Hamming	2.19	4.05	8.79	178
Paranoid	3.54	7.47	15.06	178
P2	3.34	4.6	16.87	689
PageRank	4.94	4.1	8.94	181

Table 2. Metrics of the size of the symmetric difference (SD) between generations. This includes the mean time to find a solution for the different minimalities.

Paranoid clearly has the worst average size of change, as is shown in table 2. This is probably an effect due to it tolerating changing component versions, therefore noise is created when versions of component are swapped randomly. This effects the individual generations change, but as can be seen in figure 1, it still closely resembles the Hamming minimality.

PageRank and Hamming have very similar results, as can be seen in table 2 and figure 1. With the fact that PageRank is twice as slow at finding solutions than Hamming, this leads us to the conclusion that a complex ranking of components may not be necessary for this problem and the greedy Hamming minimality is sufficient.

The effect of PageRank is most noticeable during user requests that cause significant system changes, because there are more decisions where it can differ from Hamming. This experiment looks at an overall view of evolving component systems, where simple solutions work for the majority of changes, however it may be that given a specific requests where large changes occur, complex criteria are preferable.

The P2 minimality causes many significant events during the initial generations of its evolution, as is seen in figure 1. This is likely due to it aggressively updating components as well as resolving user requests. However, after these components have been updated, the system reflects the Hamming minimality values. Whether or not this initial updating reduces change in the future systems is left for future research.

5 Related Work and Conclusions

Researchers often look at the mechanisms of component evolution [11], how to add or remove a changed component in a system. However we are looking at the architectural level, into the properties of the component system as it evolves. This paper has many similarities to the empirical work into software evolution by Lehman [12], which looks for laws and relationships between software and its evolutionary process. In this paper we aimed to look into the evolution of component systems through CDR, and identify the consequences of different definitions of minimal change.

Research into how the finer grained evolution of individual components integrates with the larger architectural view of component systems usually focuses on component versioning [13, 14]. This work aims to add well defined semantics to component versions, allowing them to describe if components has changed its external requirements or if it has only changed internally. Adding this to component models would allow CDR to make more informed decisions during evolution, and through experiments similar to the one described in this paper, their semantics could be tested to find if their effect is positive to system evolution.

As mentioned many times throughout this paper, the Mancoosi organisation is producing significant research in the area of Component Dependency Resolution. They have defined the format CUDF [10], to express and solve the CDR problems. They also run constant comparisons against different CDR implementations through the Mancoosi International Solver Competition (MISC), which we based our experiment on. They also contribute significantly in the areas of modelling the problem [15] and defining and researching solutions for it [16].

Belief revision is an area of logic that focuses on changing beliefs to take into account new knowledge. The Alchourrón, Gärdenfors, and Makinson (AGM) [17] postulates give a set of eight properties that a revision operator needs to satisfy in order to be considered rational. An intention of these postulates to ensure minimal change of the set of knowledge when it is revised. For example, if you saw what you thought was a flying pig, but you later found out it was a bird; when using belief revision you should revise the minimal amount of knowledge, therefore only discard the belief it was a pig, while maintaining the belief it was flying. By stating whether a component is installed or not as a fact, and revising these facts with user requests, we are able to define CDR w.r.t. belief revision. This area of logic is very focused on what is minimal and how to define a minimal operator that is considered rational; we have taken much inspiration from this domain.

5.1 Future Work

A drawback of our experiment is that it is not entirely representative of the CDR problem it models. The set of components should increase and change over time, and the user requests should include more than installing a single component.

The set of components over the course of evolution will change given external developers are continually adding newer versions of already included components, or new components are added. This could be included into the experiment by using past versions of a repository to simulate these changes.

The user interactions with current CDR implementations are more diverse than adding a single component, a user may also want to remove, or update a component, or change or update a repository. These interactions could be implemented within our experiments, and simulated at intervals that approximately model a real users interaction. This would increase the applicability of our results to real solvers, therefore is a key future improvement.

This work could also be generalised by experimenting on other component models instead of just Ubuntu. The goal of these experiments would be to show invariants and differences between component model evolution. This will soon be possible as the CUDF specification is being ported to other component models like OSGi. Using the OSGi based Eclipse IDE as a base system, and running the same experiments we could further this research into component system evolution.

5.2 Conclusion

In this paper we first discuss and define component system evolution w.r.t. minimal change and the CDR function. We then define four minimal change operators and use these in an experiment to find their properties over system evolution. General conclusions from this research are:

– There is a high probability a set of user requests is not satisfiable
– Most user requests cause little change, however a few requests require significant change to the system
– Complex definitions of minimality (like PageRank) have little difference from greedy simple definitions (like Hamming) and take longer to find a solution

These points need further investigation to further the knowledge about the effects of CDR on evolution of component-based systems.

References

1. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. 2nd edn. The Component Software Series. Addison-Wesley Longman Publishing Co., Boston, MA, USA (2002)
2. Jenson, G., Dietrich, J., Guesgen, H.W.: An Empirical Study of the Component Dependency Resolution Search Space. In: Proc. CBSE'2010, Springer (2010) 182–199
3. Porter, B., Sanchez, C., Casey, J., Massol, V.: Better Builds with Maven. MaestroDev (2008)
4. Rapicault, P., Le Berre, D.: Dependency Management for the Eclipse Ecosystem: An Update. In: Proc. LaSh'2010. (2010)
5. Savolainen, J., Oliver, I., Myllarniemi, V., Mannisto, T.: Analyzing and Restructuring Product Line Dependencies. In: Proc. COMPSAC'2007. Volume 1., IEEE Press (2007) 569–574
6. Barth, A., Carlo, A.D., Hertzog, R., Schwarz, C.: Debian developer's reference (2005)
7. Hamming, R.: Error detecting and error correcting codes. Bell System Technical Journal **29**(2) (1950) 147–160
8. Rapicault, P., Le Berre, D.: Dependency Management for the Eclipse Ecosystem. In: Proc. IWOCE'2009, ACM (2009)
9. White, S., Smyth, P.: Algorithms for estimating relative importance in networks. In: KDD'2003, New York, New York, USA, ACM Press (2003) 266 –275
10. Treinen, R., Zacchiroli, S.: Common upgradeability description format (CUDF) 2.0 (2009)
11. Wang, X., Mei, H., Shen, J., Wang, Q.: A component-based approach to online software evolution: Research Articles. J. Softw. Maint. Evol. **18**(3) (2006) 181–205
12. Lehman, M.: Programs, life cycles, and laws of software evolution. Proc. IEEE **68**(9) (1980) 1060–1076
13. Bauml, J., Brada, P.: Automated Versioning in OSGi: A Mechanism for Component Software Consistency Guarantee. In: SEAA'2009. (2009) 428–435
14. Stuckenholz, A.: Component evolution and versioning state of the art. SIGSOFT Softw. Eng. Notes **30**(1) (2005) 7
15. Di Cosmo, R., Zacchiroli, S.: Feature Diagrams as Package Dependencies. In: Software Product Lines: Going Beyond. (2010) 476–480
16. Trezentos, P., Lynce, I., Oliveira, A.L.: Apt-pbo: solving the software dependency problem using pseudo-boolean optimization. In: ASE'2010. (2010)
17. Alchourrón, C., Gärdenfors, P., Makinson, D.: On the logic of theory change: Partial meet contraction and revision functions. Journal of symbolic logic **50**(2) (1985) 510–530