Detecting Structural Changes in Object Oriented Software Systems

Rajesh Vasa, Jean-Guy Schneider, Clinton Woodward and Andrew Cain
Swinburne University of Technology, Melbourne, Australia
rvasa@swin.edu.au, jschneider@swin.edu.au, cwoodward@swin.edu.au, acain@swin.edu.au

Abstract– **It is an increasingly accepted fact that software development is a non-linear activity with inherently feedback driven processes. In such a development environment, however, it is important that major structural changes in the design and/or architecture of a software system under development are introduced with care and documented accordingly. In order to give developers appropriate tools that can identify such changes, we need to have a good understanding how software systems evolve over time so that evolutionary anomalies can be automatically detected. In this paper, we present recurring high-level structural and evolutionary patterns that we have observed in a number of public-domain object-oriented software systems and define a simple predictive model that can aid developers in detecting structural changes and, as a consequence, improve the underlying development processes.**

I. Introduction

The field of Software Engineering has developed and refined a number of techniques, as well as principles, that guide the development of high quality software systems. In contrast to a number of other engineering disciplines, it is commonly accepted in this field that the development of software is a non-linear activity with inherently feedback driven processes [21] and that once-off software systems are more the exception than the rule [17]. In essence, starting from an initial set of requirements, most software systems evolve over a number of releases, each new release involving the following activities: (i) defect identification/repair, (ii) addition of new functionality, (iii) removal of existing functionality, and (iv) optimisations/refactoring. When looking at this process from an evolutionary perspective, software developers tend to undertake all of the activities outlined above between two releases of a software system, possibly resulting in a substantial number of changes to the original system.

Within an evolutionary development environment, it is important that major changes in the design and/or architecture of software systems are introduced with care and documented accordingly. Despite a number of improvements in this area, research in the field of software evolution suggests that for systems to evolve, key decisions regarding design need to be rationalised and effectively communicated to the development team [1][5][16]. Hence, there is a clear need to give software developers a set of techniques that are able to automatically detect major design and architectural changes of software systems so that such changes can be documented accordingly, facilitating further evolution of the these systems. But what kind of techniques do we need?

In order to answer this question, we need to get a better understanding of how software systems evolve over time. As an important step into this direction, we have investigated structural and evolutionary patterns in object-oriented software system. More precisely, we have been observing a number of object-oriented software systems from different views to gain a better insight into the nature and types of changes as well as any clear and interesting structural patterns. As a starting point of our work, we analysed the relationship between the total number of classes and the various relationships that these classes have in a number of public-domain object-oriented software system.

The eventual goal of our efforts is to help development teams improve product quality and product documentation by using the knowledge gained by this type of research. Inspiration for our study has been taken from similar work carried in other engineering disciplines [13][14][18] as we were interested if the findings of common patterns in complex systems, like electricity grids [20] and neural networks, would also hold in software systems.

In this paper, we present our initial findings that highlight certain recurring high-level "structural and evolutionary patterns" that we have observed in a number of object-oriented software systems. Based on our observations, we have defined a simple model that can be used to determine specific attributes of a software system in future versions. Using this model, along with the understanding of the observed patterns, it is possible to identify significant design and architectural changes automatically, giving the development team information about worthwhile evolutionary trends in the software system that they are working on.

This paper has been structured as follows: Section II provides an overview of software dependency graphs which form the input for our analysis. In Section III, we illustrate our selection methodology as well as the set of techniques used to extract information from the software systems under investigation. Section IV details our observations and discusses findings which led to the development of the link growth estimation model presented in Section V, followed by a discussion of related work in Section VI. We conclude this paper in Section VII with a summary of the main insights as well as open areas for further work.

0-7803-9508-5/05/$20.00 ©2005 IEEE

II. Software dependency graphs

Graph theory can be used to define a graph G as a set of nodes V and links E. An object oriented software system can be represented as a *directed* dependency graph $G(V,E)$, where all *types* (i.e. classes and interfaces) can be treated as nodes n (where, $n \in V$). The relationships between these *types* form the directed links l (where, $l \in E$) in the graph. Throughout the rest of the paper, N denotes the number of nodes (i.e. $N = |V|$) and L denotes the number of links (i.e. $L = |E|$) of a given graph G.

Given any node n in the graph, we can measure its in-degree as well as the out-degree. The in-degree $l_{in}(n)$ is the number of inbound links into n, whereas the out-degree $l_{out}(n)$ is the number of outbound links from the node n. The in-degree can be seen as a measure of the popularity of a node in the graph, whereas the out-degree is a node's usage of other nodes in the software graph [15]. Furthermore, we define the total number of in-degree links as $L_{in} = \sum l_{in}(n)$ for all $n \in N$. Similarly we define the total number of out-degree links as $L_{out} = \sum l_{out}(n)$ for all $n \in N$.

Dependency graphs can be generated by extracting the static relationships between *types* (i.e. classes or interfaces) or we can create the graph by analysing the dynamic run-time dependencies. Static dependency graphs can be extracted from code, however to generate a dynamic dependency graph a snap-shot needs to be taken of a system during runtime.

Once a software system has been represented as a dependency graph, we can apply graph theoretical approaches to observe and analyse how these systems evolve. A set of dependency graphs can represent different versions of a given software system.

III. Methodology

A. Objective

The objective of our research is to gain a better insight into the inherent nature of object oriented software systems as well as how these systems evolve. To this end, we analyse a number of software systems over a period of time and catalog global structural patterns. Further, based on these patterns, we define a simple technique that can provide developers feedback about their software system. The motivation for our work came from prior research into complex systems which have identified structural patterns in biological systems [8], the World Wide Web [14] as well as software systems [13][18][19].

To help focus our efforts, this study starts by validating certain structural relationships identified in earlier work [13] [19] which indicates the existence of a power scaling relationship between the number of nodes and the number of edges in the static dependency graph of a software system.

B. Input data set selection methodology

For the purpose of our empirical study we restricted our input to open source software systems developed using the Java programming language [4]. The rationale for using open source software is the ready availability of a number of versions, access to detailed change logs (such as developer release notes or CVS logs), as well as non-restrictive licensing that allows us to analyse the code freely. The choice of programming language is limited by the tools developed to perform data extraction, and our interest in understanding software developed using Java as it is a popular language that has gained widespread usage in a number of different domains.

TABLE 1

Summary details of the software systems selected. N_1 is the number of nodes in the initial version and N_f is the number of nodes in the final version. Rel. is the number of release builds, Months is the duration (measured in months) of evolution.

System Name	Category	N_1	N_f	Rel.	Months	URL
Azureus	Application	103	1913	15	19	http://azureus.sf.net
Check Style	Application	25	238	12	37	http://checkstyle.sf.net
JabRef	Application	93	803	11	17	http://jabref.sf.net
JChem Paint	Application	2815	3647	19	20	http://jchempaint.sf.net
Ant	Application	70	380	14	56	http://ant.apache.org
PMD	Application	125	364	35	35	http://pmd.sf.net
Jung	Library	121	342	15	21	http://jung.sf.net
Axis	Library	166	367	21	45	http://ws.apache.org/axis/
Jasper Reports	Library	88	342	27	42	http://jasperreports.sf.net
iText	Library	184	399	25	38	http://itext.sf.net
Data Vision	Library	381	1130	58	40	http://datavision.sf.net
Saxon	Library	407	643	17	43	http://saxon.sf.net
Spring	Framework	431	1031	19	20	http://www.springframework.org
Flow4J	Framework	130	272	42	12	http://flow4j.sf.net
Hibernate	Framework	120	805	27	40	http://www.hibernate.org
WebWork	Framework	75	210	10	16	http://www.opensymphony.com

In order to select a set of software systems for our study, we defined a number of selection criteria, taking into consideration complexity, size, evolution history, recent development activity and popularity of the software system. The size and skill of development teams was not taken into consideration.

Our selection criteria are defined below. The systems have:

- At least 10 release builds available. We select a build, as released by the development team. Branch builds are ignored and only the builds created from the main tree are considered.
- Been in active development for at least 12 months to increase the likelihood of significant development history.
- Reached 200 classes (or above) at some time during its evolution and has a minimum of 150 classes by the last version under analysis.
- Detailed change logs provided by the development team as this information assists in attributing notable structural changes made by the development team.
- Been classified as an application, library or framework. This restriction helps ensure that we select systems that can be studied in isolation.
- An active user community. This is determined by the availability of articles (in trade publications and online magazines), books, messages on discussion boards and ranking based on the number of downloads.

Instead of using release dates or version numbers as identifiers for each version, we have used the pseudo-time measure of Release Sequence Number (RSN) [2]. The RSN is a sequential number allocated to the system based on its release date where the first version is 1 and then each subsequent version increases by one.

Using the selection criteria, our initial search resulted in the identification of over 100 software systems[1]. Time and resource constraints restricted this investigation to a subset of 16 systems (Table 1). Our final data set contains 16 systems, 367 unique versions and nearly 12,500 classes/interfaces (in total over all the various systems).

Many of the systems selected are widely used in a number of commercial projects. For example Hibernate, which is one of the frameworks that we studied, is very popular and is used in a number of commercial projects[2].

C. Dependency graph extraction

Dependency information is extracted from Java bytecode to create static dependency graphs. Since our input data set was restricted to systems developed using Java we have taken into consideration certain language specific aspects. Relationships between nodes are treated as directed links, which some previous studies of software graphs have simply treaded as undirected [18][19]. In our study, the *nature* of the relationship has been intentionally ignored and is an area for future work. Specifically, *associations* and *type/interface inheritance* are all treated as a dependency relationship. Further, we count the relationships between any two given nodes only once[3].

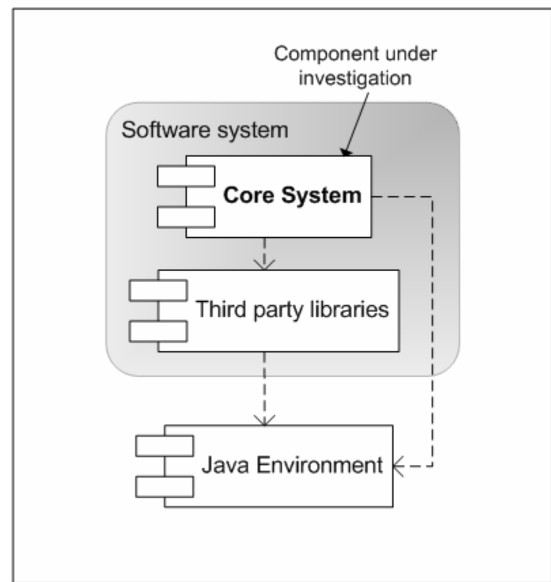

Fig 1. UML component diagram of a software system under observation. Only the Core System JAR component is investigated and used for the construction of the dependency graph.

We developed a dependency graph extraction tool that takes into consideration the language and study specific issues. This tool uses, as input, the core JAR files (Java Archive, which consist of all of the compiled source code for the system) for each version of the system under investigation (Fig 1). The tool produces a separate dependency graph for each version of the specific software system provided as input.

The extraction tool uses the Java Byte-Code Engineering library (BCEL - http://jakarta.apache.org/bcel) to collect static dependency information from the classes contained within the core system JAR. For each class, the set of dependencies are extracted and recorded. Additionally, the extraction process ignores the following dependencies:

1. The primitive types in Java[4]
2. The String class (provided by the system library)
3. The root class `java.lang.Object`
4. Self-references
5. Dependencies to inner-classes

Dependencies listed in points 1 to 4 were ignored as they form a fundamental part of the Java programming language, and therefore do not add any specific value to the analysis. The dependencies to inner-classes are treated as part of the same level abstraction, and therefore are categorised as self references.

---

[1] We manually scanned the Apache Jakarta project [*http://jakarta.apache.org*] as well as the Source Forge repository [*http://sourceforge.org*] for candidate software systems.

[2] JIRA is a commercial issue tracking system that uses Hibernate, see *http://www.atlassian.com/jira*

[3] For example, if *class A* depends on 3 methods in *class B*, we count that as one directed dependency link between *class A* and *class B*.

[4] Java has the following primitive types: `char`, `byte`, `short`, `int`, `long`, `float`, `double` and `boolean`.

It is worth noting that all of the systems under investigation have a core JAR file with optional additional third party libraries, all of which rely upon the Java environment (Fig 1). Since our study focuses on observing the evolution of the component under investigation, the internal structures of the third party libraries, as well as the standard Java libraries, are ignored by the extraction tool. This decision was made to ensure that changes within the associated libraries do not impact upon the analysis of the component under investigation.

IV. Observations and Analysis

A. Relationships between nodes and links

In their work on software architecture, Valverde et al. [19] put forward a power scaling relationship (1) between the number of nodes N and the number of links L in a software dependency graph.

$$L \sim N^{\beta}, \text{ where } \beta = 1.17 \quad (1)$$

As stated, we started our analysis to verify this relationship with our data. Since our analysis is based on a *directed* graph, we differentiate the in-degree and out-degree links. Due to the availability of this additional information we have been able to make observations for both the in-degree scaling exponent β_{in} and the out-degree scaling exponent β_{out} (2).

$$\beta_{in} = \frac{\ln(L_{in})}{\ln(N)}, \beta_{out} = \frac{\ln(L_{out})}{\ln(N)} \quad (2)$$

Although our observations support the existence of the power scaling relationships ($L \sim N^{\beta}$), we found that both β_{in} and β_{out} are not constant values and that there are significant changes between different software systems (Fig 2).

B. Bounded scaling exponents

Our observations indicate that both β_{in} and β_{out} are tightly bounded. However, they are not constant across different software systems nor are they constant across different versions of the same software system. Further, our analysis shows that β_{out} is in the range 1.25 – 1.70, while β_{in} in the range 1.15 – 1.36.

The scaling exponents for both the in-degree (β_{in}) and out-degree (β_{out}) change over the releases for all of the software systems under investigation. An interesting observation is that the amount of change between any two versions is very small and our analysis indicates that this variation does not strongly correlate with the variation in the total number of nodes.

C. Scaling exponent trends

A notable observation is that for systems which have their out-degree scaling exponent at the higher-end of the range (i.e. close to 1.70), the exponent value tends to decrease towards the lower end of the range to around 1.3. On the other hand, systems that start at the lower-end of the identified range (i.e. close to 1.25) tend to have a scaling exponent that is either very stable or only slightly increases. This suggests that most systems over time will tend to have a β_{out} in the range of 1.25 – 1.40.

β_{in} behaves slightly differently to β_{out} in that it has a much smaller range of variance. Further, this variability reduces with the maturity of the software system and tends to be lower in the later versions of the software system. This suggests that as software systems mature, the overall internal structure is resistant to significant changes.

D. Outliers systems

There is one significant outlier system (WebWork) in our data that had β_{in} in the range of 1.05 – 1.07. Upon further investigation it was found that the method used for data collection by our tool was unable to detect relationships specified through external configuration files. A more

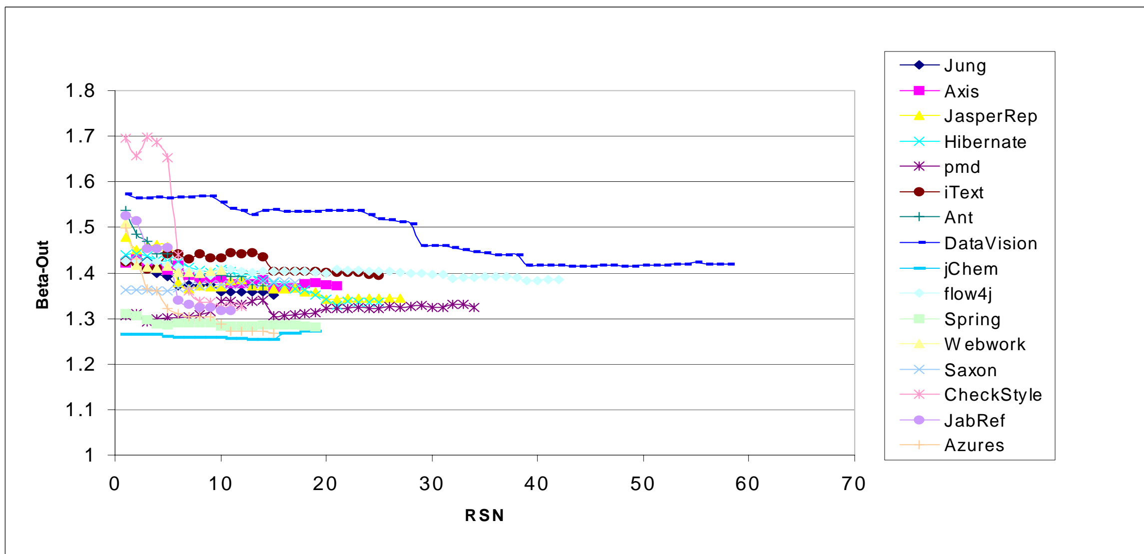

Fig 2. Value of β_{out} for the 16 systems across various versions.

detailed analysis revealed the use of the Dependency injection pattern [3] which causes this unusually low β_{in}.

This discovery prompted investigation to see if other systems at the lower end of β_{in} range also made use of the same pattern. We identified only the Spring Framework system which makes limited use of the same pattern for configuration, having β_{in} in the range of 1.141 – 1.147.

Based on these two examples we postulate that most systems will tend to have β_{in} over 1.15 and if β_{in} is lower it indicates the use of the Dependency injection pattern, or a similar pattern.

E. Discussion

The strongly constrained values of β_{in} and β_{out} indicate that the growth rate of software systems, in terms of their dependencies, is bounded and the growth of the total number of links is predictable. Based on our observations, we postulate that object oriented software systems would have their β_{in} and β_{out} values in the ranges of 1.15 – 1.36 and 1.25 – 1.70, respectively. When values are out of this range, a closer examination of the product and the process is warranted.

A high value of β_{out} would suggest that the development team might lose familiarity with their software system and eventually will find it harder to grow the software system. A high or a positive slope for β_{in} over time can be seen to be an indication of increase in coupling at the systemic level, which would also hinder growth and maintainability of the software system.

We would like to highlight one system, the Azureus bit-torrent client. It increased in size over a period of 1½ years from 103 nodes to 1913 nodes; nearly 18 times its initial size. Despite this large relative increase in size, the β_{in} and β_{out} values remained within the bounds stated and variation between two consecutive versions is very small. However, both the β values have a negative slope which indicates that the total number of new connections that nodes can support decreases as a system evolves and eventually will reach the lower end of our range, where the growth rates are near linear.

As the number of links in a system can be seen as a broad measure of structural complexity, our observations suggest that the growth of structural complexity is highly predictable. Further, since all of the observations were taken from systems that have exhibited evolution over a period of time, we hypothesise that a low variability in the scaling exponents over a significant period time is a requirement for sustainable evolution.

V. Detecting Structural Change

A. Model for estimating link growth

We observed a pattern of small change in the value of the scaling exponent β between two successive versions (in a given system). Based on this, we define a simple model to estimate the expected number of links (L_{out} and L_{in}) based on the knowledge of the β_{in} and β_{out} values from previous versions. The rationale for this is that we should be able to estimate the number of links based on the scaling exponents from previous versions. When the estimated number of links is significantly different from the observed number of links, it indicates a notable structural change in the system.

Our model (3) includes the following observations and considerations:

1. Scaling exponent does not change significantly between any two given versions.
2. Magnitude of size change (node count) between two versions does have an impact on the scaling exponent, but the magnitude of such impact is unknown.
3. Maturity and size of a software system influences the amount of expected variation.
4. Model is applied to directed graphs, and so can be used to predict either in-degree or out-degree connectivity.

The estimation model for the number of system links in a given version v is L', where L' denotes either the estimated number of in-degrees or out-degrees. This is based on then known number of nodes N_v at version v, and is scaled by the estimated scaling exponent β'. The estimation model is shown in (3).

$$L' = N_v^{\beta'} \tag{3}$$

The scaling factor β' is treated as a dynamic function that takes into consideration the scaling exponent β_{v-1} value from the previous version $v-1$ and a relative measure of system size changes Δs, as shown in (4).

$$\beta' = \beta_{v-1} - p\left(\frac{\Delta s}{\ln(N_v)}\right) \quad \text{where,} \quad \Delta s = \frac{N_v - N_{v-1}}{N_v} \tag{4}$$

A constant probability multiplier p limits the effects of relative system growth.

B. Estimating links in the data set

We applied the model to our input data set where the p value was initially set to *zero*. This model was able to estimate L_{out} to within a 7% threshold error margin (t) for 90% of the software systems (Fig. 3). However it was more accurate calculating L_{in}, estimating 92% of the versions within 7% error margin.

Our analysis indicates that p can be set globally across all systems to increase the accuracy of the model. When a constant value of 0.55 was supplied to estimate L_{out}, the model increases its estimation accuracy to nearly 95% with the same 7% threshold value for error, similarly a value of 0.2 when estimating L_{in} increased the model accuracy to 95%. We hypothesise that p is a system dependent parameter and for optimal performance it should be adjusted to accommodate system specific qualities. Preliminary work suggests that adjusting p for the various maturity stages of a system is ideal.

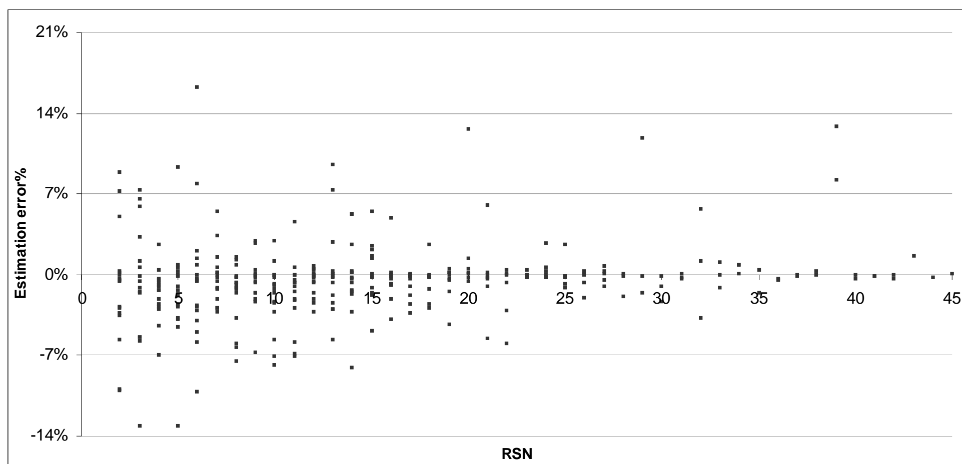

Fig 3. Scatter plot of estimation error across 16 systems when calculating L_{out}.

C. Identifying structural change

The full benefit of our model is visible when we start to look at points in the data where the error is above the threshold value. An investigation of some of the instances when the error rate of the estimation model exceeds *t* indicates that the systems have undergone significant change.

Due to the quality of information in change logs, we were unable to attribute a cause for the changes in all instances. However, for nearly 80% of instances where the data was over the threshold value we were able to attribute a cause by analysing the change logs. Our analysis involved reading the change log provided to look at potential factors indicated by the development team.

When we summarised the factors across a number of systems we identified certain recurring themes:

- Substantial refactoring of the system. In most of these cases, the L_{in} value changed significantly, the size of the system exhibits little to no change.
- Work done to improve performance. (e.g., extensive use of new algorithms or data structures to improve performance of a system). Depending on how these were achieved, either the L_{in} or the L_{out} value is outside of the expected range.
- Changes to the architecture of the system. (e.g., moving to a plug-in based architecture for extensibility). Generally in these cases, the estimated L_{in} value is outside of the expected range.
- Extensive use of a third-party libraries/component. (e.g., use of XML libraries to provide configuration). The estimated L_{out} value typically is the one that exhibits a large error in these cases.
- Work done to improve the API, typically interfaces provided to other developers to extend and use the product. These involve changing core components and hence the system structure changes.
- Changes to the way the exceptions and errors are handled in the system. These changes are system wide and hence we would expect a significant impact at the system level.

Our observations lead us to postulate that when the error margin is over a certain threshold *t,* it indicates a change in the system that warrants investigation and suitable documentation to justify this deviation from estimated values. The threshold value *t* should be system specific and will need to be adjusted by the development team based on their knowledge of the processes guiding the evolution of the software system. We support our recommendation with the observation that for some systems (Hibernate, Spring Framework, JChemPaint etc.) the estimation error was never over the threshold value we defined at 7%. This indicates that for these systems a lower value might identify aspects that the development team find valuable. Although the depth of our analysis does not allows us to provide a recommended range for the threshold *t*, it would be recommended not to use too low a value as normal development tasks might be picked up by the model.

We would like to stress out that our model is suited to detecting structural changes that have an impact on large parts of the system. Such changes are worth documenting or investigating. However, the model is limited in that it is unable to detect all significant changes. Hence, this model

TABLE 2.
Subset of estimation errors and reasons identified. Versions are the successive version numbers. Cat. indicates the link category. Actual and Estimate show the link counts, and Err% is the relative error between the actual and estimated values.

System	Versions	Cat.	Actual	Estimate	Err%	Reason
JabRef	1.3.1 – 1.4	L_{out}	3308	3568	7.9%	Extensive use of new parser
PMD	1.01 – 1.02	L_{in}	1246	1439	15.5%	Different exception propagation strategy
Ant	1.6.1 – 1.6.2	L_{in}	1331	1563	17.4%	Exception mechanism changed
Flow4j	0.8.8 – 0.9	L_{out}	2230	2523	8.3%	Extensive use of new libraries
Check Style	2.4 – 3.0	L_{in}	277	383	38.2%	New plug-in based architecture
Data Vision	0.8.4 – 0.9	L_{in}	970	1064	9.7%	Refactoring

should be used to complement the existing development practices to provide the team with feedback and information that would warrant retrospection. In most cases it would be logical to assume that the development team are aware of the causes of the changes, and hence the models true benefit would be for managing the development effort and to ensure that a suitable documentation trail has been provided to justify the estimation errors over the set threshold.

VI. Related work

Motivated by research in complex systems, Valverde et al. [19] studied a number of different software systems to see if they exhibited any emergent properties at the system level. They focused their efforts on structural properties of static software dependency graphs and found certain common properties, such as a high clustering coefficient, and small-world like networks based on the short path lengths, which are relatively invariant across the 29 different systems they investigated. In their work they ignore the directed nature of software dependency graphs. Jenkins et al. [7] extended the work done by Valverde and studied the evolution of the core Java libraries based on static directed dependency graph analysis. They conclude that software systems have a scale-free character, which is a strong indicator of small-world like networks. More recently, Potanin and their team [15] suggested that object oriented software systems have scale-free properties; their conclusions are based on the analysis of dynamic dependency graphs.

Much of the seminal work in the field of software evolution has been done over a number of years by Lehman et al. [9] [10], putting forward 8 laws of software evolution. Their work suggests that at the system level the evolutionary behaviour is systemic and not completely under the control of the individual developers. Essentially, they see this as an emergent property of the development process as well as the system. Kemerer and Slaughter [6] reviewed existing literature on software evolution and have designed an approach for longitudinal research that enlarges the scope of empirical data available on software evolution. They claim that "software evolution patterns could be examined across multiple levels of analysis (system and module), over longer periods of time, and could be linked to a number of organisational and software engineering factors". Mockus et al. [12] studied the evolution of Mozilla and the Apache web server, both open source software systems. They argue that the design structure of the software system has a direct impact on the development speed; a highly modular, component based architecture allows fast evolution whereas a highly interdependent architecture generally requires a longer period of time between released versions.

None of these approaches, however, attempt to exploit the emergent structural anomalies in evolving systems to provide feedback to developers writing code.

VII. Conclusions and further work

Modern software systems are complex and made up of a large number of interacting parts. Complex systems theory suggests that when studying such systems it is important to infer from the systems itself how it can be best described rather than to focus on individual parts [11]. This suggestion is based on prior research efforts into complex systems [14] that indicate that complex systems exhibit a certain set of emergent properties, i.e. attributes that are visible only at the system level and not necessarily intentionally created by the developers of these systems. Studies into such emergent properties can provide a valuable insight and a different dimension to our understanding of software systems. Motivated by work in complex systems, we set out to identify structural patterns in object oriented software systems and how an understanding of this can help development teams improve their processes.

In this work, we presented a common structural pattern across a number of different software systems. More specifically, we illustrated the bounded and predictable power-relationship between the total number of nodes and the total number of links in a static dependency graph representation of an object oriented software system. Based on this relationship, we defined a simple model that is able to estimate the growth of the total number of links in a software system. Finally, we have proposed a technique that applies the model to detect significant structural changes between two consecutive versions of a software system.

When used as part of the development process, our technique can be used by a team of developers to raise questions, which when answered, will leave behind a valuable trail of information that will help to better document the decision history of a project.

A key limitation of the work arises from the size of the data. Similarly, the entire data consists of open source software systems developed using the Java programming language. This limits certainty when generalizing our findings to systems that are outside of the constraints. However from a theoretical perspective, we postulate that the range of values identified will hold for other systems developed using any object oriented programming language as values outside of suggested boundaries would imply a significantly unusual structure that would constrain the growth of the software system.

As we continue research in this area it is desirable to validate our findings against more systems, including those developed using other languages, as well as systems that are commercial and closed source, to see if there are significant differences to the boundaries that we have identified for the β_{in} and β_{out} values. We also believe that further analysis of dependency graphs by considering specific types of relationships only (e.g. inheritance) may provide greater insight into other properties and patterns. Such information should help developers to better understand the dynamics of software evolution and emergent properties of the software systems as they are being produced.

REFERENCES

[1] J. Bosch, "Software Architecture: The Next Step," European Workshop on Software Architecture, 2004.

[2] D. R. Cox and P. Lewis, "The Statistical Analysis of Series of Events", Methuen, London, 1966.

[3] M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern", http://martinfowler.com/articles/injection.html, 2004.

[4] J. Gosling, B. Joy, G. Steele and G. Bracha, "The Java Language Specification", Second edition, Addison-Wesley, 2000.

[5] C. F. Kemerer, "Empirical Research on Software Complexity and Software Maintenance", *Annals of Software Engineering,* 1(1): 1-22, 1995.

[6] C. F. Kemerer and S. Slaughter, "An Empirical Approach to Studying Software Evolution," *IEEE Trans. Software Eng.*, 25(4): 493-509, 1999.

[7] S. Jenkins and S. R. Kirk, "Prediction of software evolution as a complex network", submitted to Journal of Information Science, 2004.

[8] H. Jeong, B. Tombor, R. Albert, Z. N. Oltvai, and A. L. Barab´asi, "The large-scale organization of metabolic networks", *Nature* 407, pp. 651–654, 2000.

[9] M. M. Lehman, "Programs, Life Cycles, and Laws of Software Evolution," In Proceeding of Special Issue Software Eng., IEEE, 68(9): 1060-1076, 1980.

[10] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski. "Metrics and laws of software evolution – the nineties view." In Proceedings of the Fourth International Software Metrics Symposium (Metrics'97), 1997.

[11] M. Mitchell and M. Newman. "Complex Systems Theory and Evolution," in Encyclopaedia of Evolution, M. Pagel (editor), Oxford University Press, 2001.

[12] A. Mockus, R. T. Fielding and J. D. Herbsleb. "Two Case Studies of Open Source Software Development: Apache and Mozilla." *ACM Transactions on Software Engineering and Methodology*, 11(3): 1-38, July 2002.

[13] C. R. Myers, "Software systems as complex networks: structure, function, and evolvability of software collaboration graphs," *Phys. Rev. E.*, Vol. 68, 2003.

[14] M. E. Newman, "The Structure and Function of Complex Networks," *SIAM Review,* 45(167): 167-256, 2003.

[15] A. Potanin, J. Noble, M. Frean, and R. Biddle. "Scale-Free Geometry in OO Programs," *Communications of the ACM*, 48(5): 99-103, May 2005.

[16] B. Ramesh, M. Jarke, V. Informatik, and R. Aachen, "Towards Reference Models for Requirements Traceability," *IEEE Transactions on Software Engineering*, 27(1): 58-93, 2001.

[17] I. Sommerville, "Software Engineering", Seventh edition, Addison-Wesley, 2004.

[18] S. Valverde, R. Ferrer Cancho and R. V. Sole, "Scale-Free Networks from Optimal Design," *Europhysics Letters*, 60(4): 512-517, 2002.

[19] S. Valverde and R. V. Sole. "Hierarchical Small-Worlds in Software Architecture," ArXiv preprint cond-mat/0307278 - arxiv.org, 2003.

[20] D. J. Watts, Small Worlds, Princeton University Press, 1999.

[21] L. Williams and A. Cockburn. "Agile Software Development: It's about Feedback and Change," *IEEE Computer*, pp. 39-43, Jun 2003.