Characterizing the Technical Potential of a Software Module

Sushil Krishna Bajracharya and Trung Chi Ngo

Department of Informatics
Donald Bren School of Information and Computer Sciences
University of California, Irvine
{sbajrach,trungcn}@ics.uci.edu

Abstract

Net Options Value (NOV) is a quantitaive model to evaluate design options in modular systems. We present an empirical study to understand an important parameter of the NOV called the 'technical potential' of a module. First, we provide a brief introduction of the NOV model, summaries of analyses - where and how NOV has been applied, and list some of the open issues with NOV. Second, we present the empirical study, where we discuss; (i) our method for gathering required data from the CVS repository and the Bugs and Issues database of ArgoUML, the application we studied; (ii) the results and how they relate to our hypothesis about technical potential, and (iii) the current limitations of our analysis and future directions for elaborating the experiment we carried out in a limited period of time.

1. Introduction

Net Options Value (NOV) is a mathematical model for evaluating modular design structures. It is based on the economic theory of real options and was first introduced by Baldwin and Clark in [6]. The model takes design as an experiment or investment activity whose outcome is unknown. In terms of returns, it might be profitable (better design) or might incur loss (worse design). NOV models such uncertainty by assuming the expected value of a module to be a random variable with a normal distribution. It accounts for the cost involved in making design changes based on the complexity of the modules and the dependencies between the modules. The result from NOV analysis is a real valued number that is taken as an indicator of the value for a given design.

1.1 Applications of NOV

It has been demonstrated that NOV is capable of evaluating modularity in software design at varying degrees of granularity: at the level of a small application [15], moderately sized application using Aspects [11] and at a much grander industrial scale [6].

[15] used NOV in the analysis of designs for the KWIC program proposed by Dave Parnas in [12], and, using results from NOV analysis, validated that an information hiding design is superior to a *protomodular* design. [11] demonstrated how NOV applies to a real world application by using a non-trivial example that showed that aspect-oriented modularization, in certain cases, adds value to an existing modular design. Recently, NOV has been used in the evaluation of various case studies of aspect-oriented designs whose results were consistent with those established in the literature. [10] In other words, NOV results confirmed the established notions of good and bad designs using Aspects.

1.2 Open Issues in NOV

Despite these attempts and efforts that support the usability of NOV and its capability to assess design value, irrespective of granularity and implementation strategy, there are many open issues in adopting this model as a valid framework to be used in practice.

Some of the immediate questions that emanate out from these early works done with NOV are as follows:

1. Can we safely generalize the assumptions in NOV to be true for software, across domains, and other dimensions along which software design may vary?

2. How do we associate the numeric value from NOV with meaningful attributes like effort or other related estimates?

3. Is there a valid mapping between conventional software metrics/measures and the parameters in NOV?

4. How do we incorporate the differences between various forms of modular dependencies? In doing a fine grained analysis we come across with questions like "Does inheritance has the same dependency *weight* as a method call?"

5. Many implicit dependencies exist between modules, for example, social dependencies [7]. How do we incorporate such non-technical dependencies in NOV?

We need a deeper understanding of the parameters in NOV, especially the link between economic and software properties, before we are in a position to answer open questions like these. We see empirical validation of the assumptions we make about NOV and its parameters as having a great potential in answering these questions.

We argue that any further research in better understanding NOV will broadly fall into these two categories: (i) Rich representation of dependencies, and (ii) Mapping parameters with more accurately measurable software attributes. A detailed discussion of these is out of scope of this paper. Therefore, we focus on our approach to understand a more elusive parameter in NOV called the *technical potential* of a module (denoted by σ).

1.3 Technical Potential of a Module

Mathematically σ is the standard deviation coefficient in the following expression for NOV.

$$NOV_i = max_{ki}\{\sigma_i n_i^{1/2} Q(k_i) - C_i(n_i)k_i - Z_i\} \quad (1)$$

Here, $\sigma_i n_i^{1/2} Q(k_i)$ represents the expected value of the module and is a random variable with normal distribution. k is the number of experiments, n_i is the complexity of the i_{th} module, C_i is the redesign cost and Z_i is the visibility cost. Further details about the model and the parameters are given in [11, 15, 6]. $Q(k)$ is a function that models the value of best of k designs.

The NOV model treats design as an investment activity or, precisely, a *value seeking process* [6], and above expression has the following intuitive meaning.

> Net Options Value = Expected Benefit - Redesign Cost - Visibility Cost

We can directly infer only two mathematical properties of σ from the NOV model: (i) Since σ is a standard deviation coefficient in above equation, it will always be positive, and (ii) Based on the assumption that one experiment in a unimodular design breaks even [6], we get a value of 2.5 for σ of a single module system when the expected benefit and the redesign cost is assumed to be 1. These properties are not sufficient enough to make any general arguments about the exact *relations* (empirical/numerical) that hold for σ. The intuitive meaning of σ further helps in extending our understanding about it.

The intuitive meaning of σ for software was proposed in [15] where they provide the following argument.

> We have observed that the environment is what determines whether variants on a design are likely to have added value. If there is little added value to be gained by replacing a module in a given environment, no matter how complex it is, that means the module has low technical potential.

[15] associates σ with *the variation in returns* for the investment made in design. This also supports Baldwin and Clark's theory of seeing σ as the indicator of *technical risks* inherent in a module. Furthermore, [15] argued that σ depends on the environmental factors such as *computer*, *corpus* and *user*. Following the notion of *environment parameters* (factors), [11] represented a collection of web-services as *external parameters* (EPs). The authors in [11] classified the modules into three categories and assigned the maximum weight for σ if the module falled under the category that provided direct service to the end-users.

Based on these observations, both mathematical and intuitive, we argue that the right way to assess any measurement for σ is to study the evolution of existing modules. If we follow the interpretation of σ in economic terms, we would need to look into the financial history of the modules in capital markets. We believe that the history of software components in capital markets is not the sole indicator of σ, and, it is possible to look in the change patterns of modules throughout their lifetime in the light of several other environment parameters like users, external services, socio-technical implications etc.

2 Empirical Study

With the above background on NOV and *technical potential*, σ, we conducted an empirical study to validate the understanding about σ, in particular, to verify the relationship that has been assumed to hold between the external parameters and σ.

2.1 Research Hypothesis

The major hypothesis we want to test is the one that was asserted in [11], where the authors proposed a simple heuristics for σ, assuming it to be a function of the number of external parameters that a module depends on. Given below is an excerpt from [11] that explains the measurement function used to calculate σ.

> $$\sigma_i = f(e_i, p_i) = (e_i + 1) \times p_i \qquad (2)$$
>
> where e_i is the number of external parameters that the i^{th} module depends on. We add 1 to e_i as we do not want to assign σ a value of zero just because a module does not depend on the external parameters.
>
> p_i indicates the i^{th} module's relevance to end users of the system. Users directly interact with the *application controller* modules, thus we assign them a p_i value of 2. *Functional modules*, *subsidiary modules* and *design rules* are less visible to the users so we assign them a p_i of 1. We assume the value of p_i to be 0 for *web.xml* since this parameter is merely a configuration file and does not contribute much to the system's functionality and the end users.

Validating the exact mathematical function for σ as given above is not possible as the external parameters and classification of modules are specific to the application that was used in [11] and we have no historical data available for the system described there. However we seek to validate the following generally applicable assertions about σ that is the essence behind the above expression for σ.

1. σ is a function of the external parameters, meaning, if a module depends on more external parameters it should have a higher σ.

2. The value of σ differs based on the type of the module, and the services it provides.

In this paper we primarily focus on the first hypothesis. We obtained necessary empirical data to validate the above two properties about σ from the CVS repository and the defect and issue database of the application we studied. Before data collection and analysis we had to decide on defining what constitutes a module, what are the external parameters and what are the indicators of the technical potential in the historical databases available to us.

2.2 Application Studied

A major challenge in our study was that we had to pick an appropriate software project, big and mature enough, so that we could extract sufficient data points for performing the required statistical analysis. On the other hand, it had to be small enough so that we can finish our study within the ten-week time frame. Among the different choices we had, we picked ArgoUML [4] as an application to be studied.

ArgoUML is an open-source software engineering tool for object oriented analysis and design. It provides a smart and easy-to-use design environment for modelling software with UML. According to the project's website and best of our knowledge about ArgoUML, its lastest version [1] is UML 1.3 compliant. It supports all eight major UML diagrams and the UML's Object Constraint Language (OCL).

ArgoUML has its root in academic research in domain oriented software design environments. It was orginally developed and maintained by Jason Robins during his Ph.D. career in UCI. The first reference to the Argo name appeared in [13]. ArgoUML was made an open source project on February 1999 and moved to tigris.org on January 2000. It has become one of the most active open-source project at tigris.org. In 61 months from January 2000 to February 2005, the ArgoUML's development team released 44 different versions of ArgoUML.

2.3 Data Sources

We decided to gather empirical data from 28 months period, from August 2002 to December 2004. There were 29 different versions of ArgoUML released during this period. From those releases, we picked 22 versions as candidates for analysis as their official release dates [2] tend to form a normal distribution.

Two primary sources of empirical data that we investigated are:

- *CVS source repository*: The evolution of the implementation of ArgoUML is maintained with the help of Tigris's Concurrent Versions System (CVS). CVS allows tracking different versions of the entire code base through the *tag* feature. This feature allows developers to take a snapshot of the entire code base at any point in time. The snapsnot can be labelled with a unique tag for later reference and retrieval.

  The code base of ArgoUML has been controlled properly since it moved to tigris. We were able to find and retrieve different versions of ArgoUML from its CVS source repository, including the 22 versions we investigated.

- *Issuezilla bug/issue tracking system*: Issuezilla is an issue/bug tracking system provided by tigris.org. It provides a collaborative environment for dealing with issues in software development. Each project hosted by

[1] version 0.18.1a

[2] Official release dates are dates that the development team makes public announcement about the availability of new versions of ArgoUML for download. Information about these dates were acquired from the ArgoUML's website

tigris.org, including ArgoUML, has a seperate instance of issue/bug database. Five types of issues that can be reported in this database are: defects, enhancements, tasks, and patches. Depending on their permission settings, users may query, view, report, modify, and reassign issues.

In the earliest version of the ArgoUML (from the 22 versions we selected), the issuezilla database contained approximately 1.000 issues, and the last version we picked had almost 3.000 issues. With query tools provided by the Issuezilla system, we were able to download all issues reported within the inspection period.

2.4 External Parameters and indicators of Technical Potential

We selected all the major external libraries used in ArgoUML as the *external parameters* and grouped them into four major groups: (i) UML: Classes that contribute in implementing different UML features. (ii) UI: Classes that are responsible for user interface and providing the drawing capabilities for various UML diagrams (iii) Data: XML libraries that are used to handle the data, and (iv) Misc: Other miscellaneous classes that are used for internationalization, logging and language parsing.

We assumed that the number of open issues for a module in the Issues and Defects database (IssueZilla) of ArgoUML are the indicators of the *technical risk*, and thus are the indicators of the technical potential of a module in ArgoUML. We make this claim based on the mathematical fact that σ contributes to the standard deviation of the expected value of a module as defined in the NOV model.

If we look into the equation for NOV given in section 1.3, the expected value of a module is $\sigma_i n_i^{1/2} Q(k_i)$, where n is the complexity of the module and $Q(k)$ is the expected value of the best k independent trails from a standard normal distribution for all positive values in the distribution. Value of $Q(k)$ increases with k, and n and σ is what alters the expected value of the module. Since, collectively $\sigma_i n_i^{1/2}$ is the standard deviation, it implies that as n or σ increases the uncertainty about the expected value also increases. With more standrad deviation the expected value might be significantly greater or lower as compared to the case with lower standard deviation. This is just a simple probabilistic relationship that holds for a random variable with a normal distribution.

The number of open issues in the issues database of ArgoUML records defects, enchancements, feature requests, patches and tasks for several modules. A higher number of these issues recorded for a module indicates that a module is considered valuable by the users. Especially in the open source application like ArgoUML the attention that a module gets from its community is a good indicator of its usage and its value. However, the more of the entries in these issues, for example the defects and feature requests also indicate that if those defects are not fixed or features not supported in future, that might turn down number of potential users and thus degrade the value of the module for which those issues were recorded. With this argument we assume that these open issues indicate the technical risk inherent in a module. Thus, we sum up all the issues recorded for a module and treat it as the indicator of the technical risk.

With this, we need to measure the change in the number of classes from external libraries used in a module and how that relates to the change in the issues recorded for a module. Studying this change relationship will allow us to validate our hypothesis about external parameters increasing the technical potential of a module.

2.5 Logical Components from ArgoUML

After deciding that the external librarires constitute the external parameters and open issues indicate the technical potential, we had to come up with a valid definition about what constitutes a module. The straightforward way was to pick an individual Java source file or package as a module as ArgoUML is implemented in Java. This approach fails short, at least with our definition for external parameters and indicators or technical potential for ArgoUML project. It is difficult to reason about the relationship between an issue in the Issuezilla database and an individual Java source file/package. For example, if we want to tell what the Java files related to a bug report are, we would need to perform software testing for identifying and localizing the bug. This is impractical or even impossible with respect to the number of issues that we are dealing with.

This above observation led us to seek for a better mapping between issues recorded and what we would define as software modules. Fortunately, ArgoUML's issuezilla database already provides a logical grouping of related software components into 26 pre-defined modules. Thus, issue and bug reporters can specifically identify which of the predefined module(s) is related to an issue. We decided to select 4 out of 26 pre-defined modules for analysis. The remaining task was to create the mapping between Java source files and the selected modules. Based on the name and description of each pre-defined module, we were able to identify the set of corresponding Java packages. The following table shows the list of selected modules and their corresponding mapping.

Module	Java Packages
ClassDiagram	org.argouml.static_structure org.argouml.static_structure.layout org.argouml.static_structure.ui
State/Activity diagram	org.argouml.uml.diagram.state_ui org.argouml.uml.diagram.activity org.argouml.uml.diagram.activity_ui
Model	org.argouml.model org.argouml.model.uml
CodeGeneration / ReverseEngineering	org.argouml.uml.generator org.argouml.uml.generator_ui org.argouml.language org.argouml.language.helpers org.argouml.language.java org.argouml.language.java.generator org.argouml.uml.reveng org.argouml.uml.reveng.java

2.6 Data Collection

A set of small utilities was developped to automate the data collection. We wrote an Ant [3] script to perform the following tasks on each of the 22 releases we studied.

- *Downloading/compiling the source code* for each of the releases based on its corresponding CVS tag. The code was then compiled to Java bytecode for further analysis.
- *Building logical modules*: Four directories representing four logical modules described in Section 2.5 were created. Logical modules' Java .class files were then relocated to their corresponding directories.
- *Measuring dependency metrics*: The script used Dependency Finder [2], an open source tool to calculate the required dependency information we needed. For each logical module, a XML-based dependency graph representing the complete in-comming and out-going dependencies between the internal Java classes and external ones were generated. XPath [16] queries were applied on the resulted graphs to measure the the number of external Java classes used by the module.
- *Measuring the number of open issues*: The issue database was exported to XML format and downloaded manually using a web-based query tool provided by Issuezilla. The script then applied XPath queries on the downloaded XML file for measuring the number issues created during the period of the current release to the next.

2.7 Data Analysis

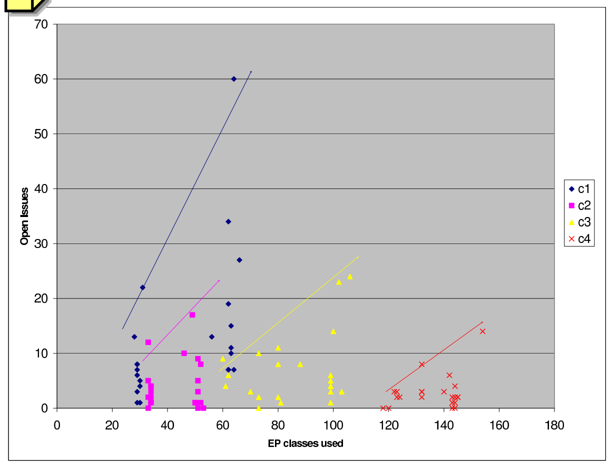
We obtained total 88 data points each with a record of number of classes used from the external libraries and the number of open issues in the issues database of ArgoUML. We had 22 data sets for each of the four logical modules we studied. Figure 1 shows various scatterplots based on these data sets. They depict the effect of increasing the number of EP classes used on the open issues recorded.

Figure 1(a) shows the plot for each of the four modules using absolute values for the EP classes and open issues. Figure 1(b) plots the same data with these values scaled to 1 as maximum for all the modules so that we can see the overall effect of EP classes on open issues. For each of the modules we can see a common pattern of higher number of issues appearing with increasing use of EP classes. However, it is clear from all the plots that the relationship between open issues and EP classes used is not strictly linear. We can see that the range, rather than the number, of open issues is increasing with the increase in the EP classes used. The common change pattern seen in all of the scatterplots is that the increasing EP classes along x-axis clusters points in y-axis such that they form a triangular shape. A directed line is drawn in the plots to highlight this triangular (or conical) shape formed by the points.

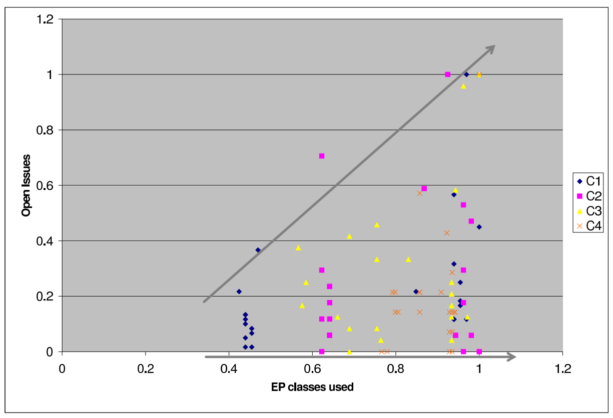
Figures 1(c) to 1(f) are the individual plots for each of the modules and they further show the effect of a module's increasing dependency with various EP classes on the open issues. These plots also show that not all EP classes are equally changing. Classes from UML and UI are the ones that are varying the most and thus we are able to observe a more noticeable variation in the issues for these EPs. For example, in the CodeGeneration module (Figure 1(e)), number of all other EP classes except UML classes are nearly fixed to a single value. Thus, the issues recorded are varying more prominently, forming a clear triangular area for the UML classes. Similarly, in module ClassDiagram (Figure 1(c), UML and UI classes collectively seem to be forming the common triangular area whereas other two EP classes, Data and Misc, are not varying along the x-axis and thus we are not able to observe their effect on issues.

2.7.1 Validating the Hypothesis

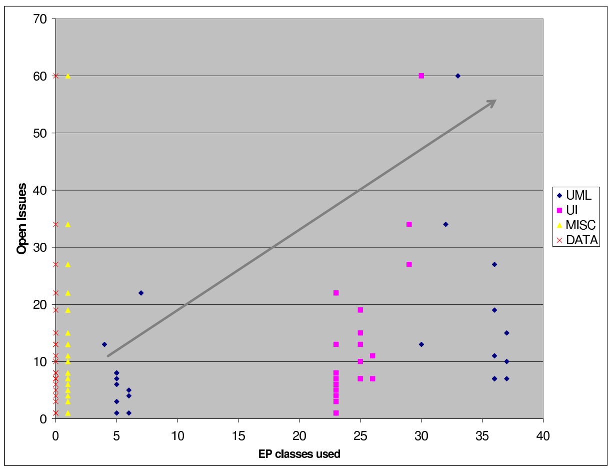
Our first hypothesis that σ is a function of the external parameters, is supported by the observed pattern in the scatterplots. From our observations made above, we see that the open issues do not just increase in their numbers, but other statistical property like the range or the deviation in the issues is increasing with increasing usage of EP classes. The number of EP classes seem to be more correlated with the deviation in open issues recorded than the actual number of issues. However, a microanalysis of this relationship



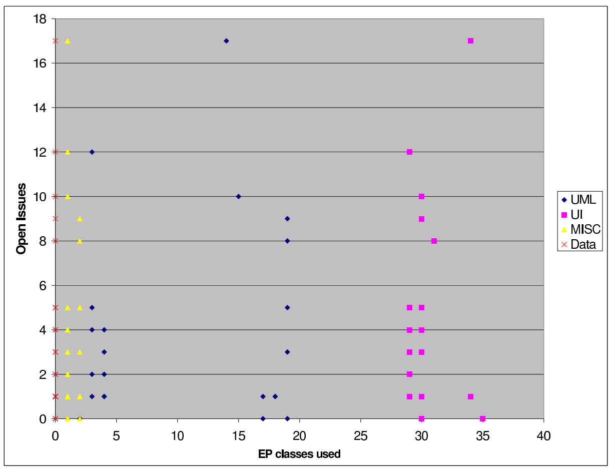
(a) Effect of EP classes used on Open Issues, modulewise



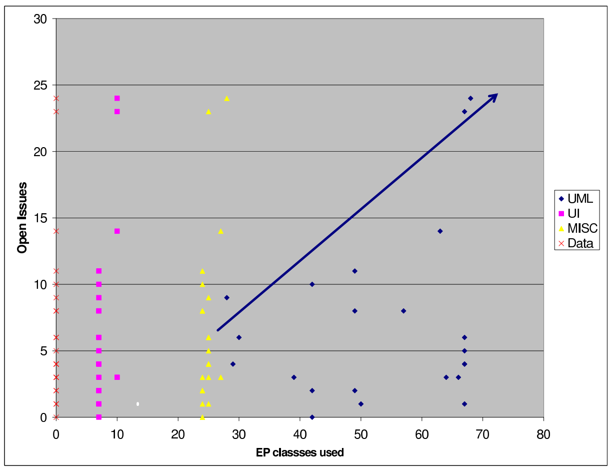
(b) Effect of EP classes used on Open Issues scaled to 1, modulewise



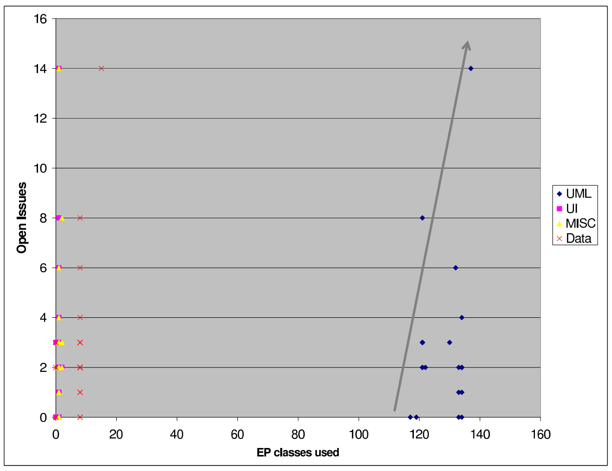
(c) EP classes used Vs Open Issues for module C1 - ClassDiagram



(d) EP classes used Vs Open Issues for module C2 - State/ActivityDiagram



(e) EP classes used Vs Open Issues for module C3 - CodeGeneration



(f) EP classes used Vs Open Issues for module C4 - Model

Figure 1. Scatterplots showing the effect of increasing use of classes from external libraries/parameters (EP) on open issues. The increasing range of issues recorded as the usage of external classes increases is shown by a straight trendline(s) that elevates up with the increasing number of EP classes used.

is not possible with the limited data we managed to collect. In general, we confirm that, a module that has more dependencies with external parameters has increased technical potential.

Our second hypothesis that, the value of σ differs based on the type of the module and the services it provides, is roughly supported by Figure 1(a). It shows there are varying degrees of increments in open issues for different modules. For example, module ClassDiagram (C1) has the highest increment in issues whereas module, Model (C4) has the lowest.

2.7.2 Threats Against Validity

As we discussed in Section 2.4 there are two factors that contributes to the standard deviation of the expected value of the module in NOV expression - Complexity and the technical potential of a module. An argument against our confirmation of the first hypothesis might be that the change in the open issues recorded was because of the change in the complexity of the modules and not directly because of the increasing depenedency of a module on external parameters. To counter this argument we need to look into the scatterplots for individual modules. Since, NOV assumes complexity as a function of the tasks performed by a module [6], we regard the complexity of a module (in its individual plot, Figure 1(c) to 1(f)) to be same throughout its 22 releases, as their major tasks and responsibilities remain same in these releases. We see that the number of open issues is changing for a module but its complexity is roughly same. Thus we confirm that the increment in open issues was, in fact, affected by a module's depenedency with the external parameters.

We cannot make the same argument to confirm the second hypothesis. In order to see the direct relationship between a module's dependency to external parameters across all four modules we need to take into account the differences between the complexities of those modules. We have not collected any data at this point that allows us to accomodate complexity of the modules in our analysis. The observation in Figure 1(a) does not refute our second hypothesis, but we defer making any further strong confirmation without a deeper analysis.

2.8 Limitations

The greatest limitation of the study we have conducted is the amount of data we collected. Since the project had to be completed in a limited amount of time and we did not have a fully automated prebuilt tool for data collection, we developed a semi-automated data collection tool/method. This limited both the dimension and amount of data we could collect. We collected data for the four modules at a fixed interval of 22 releases. We could have done a finer analysis if we had been able further quantify our data based on several other intervals of time, for example, based on the major design decisions made throughout the history of the application. Other than the conical areas formed in the scatterplots, there is one more common pattern. There seems to be a sudden leap in the number of external classes being used and thus creating vertical clusters of points in the plots with a considerable gap in between those clusters. The only exception in this case is the scatterplot for UML classes in Figure 1(e). A simple explanation for this is the fact that we collected data at a fixed interval and in many cases we observed an abrupt change in the number of classes being used from one interval to another. For example, Module ClassDiagram had used around 30 classes from the external parameter - UML in the first 11 releases and abruptly switched to using 5 classes in the next 11 releases. We could have been able to see a smoother transistion, if any, given more time and better tool to query releases at much finer detail.

Another important fact is that the application under study needs to be showing interesting evolutionary changes. The only prominent design change that we were aware of in our study was the design decision of leaving NS-UML in ArgoUML after release 0.18. [1] (This also explains some of the abrupt changes in classes from UML we mentioned above.) We could see this as the decreasing number of classes from external parameter, UML in the subsequent releases. In cases where these changes can be captured finely we see interesting results. This is also the reason why the scatterplot in Figure 1(e) shows a noticeable pattern for the UML classes from EP. Other classes that do not change do not show much useful or interesting patterns like these.

Relying on an open source project like ArgoUML has a benefit that the data we need are publicly available. But, we cannot say anything about the reliability of the data that exist in the database. Especially the date and time the issues were entered and the release numbers they relate to have a significant role in our analysis. We are not assured that these entries are always correctly made. Particularly in open source software, the history of a software module usually manifests itself as indirect evidences scattered throughout different development artifacts, including the project's documentation, repositories of source code, defect tracking databases, and mailing list archives. Although open source projects are taken as candidates systems to study software evolution [9, 14] we found ArgoUML challenging enough to use, also, due to the lack of exact set of tools we needed. A good fraction of time in the project was spent in building and integrating exisitng tools to collect data from ArgoUML. This implies that we need to devise sound tools and techniques for collecting meaningful metrics from such historic databases.

We can make two major conclusions from this experience.

1. We need to be able to pick exemplar applications for study (which is inherently difficult), and

2. There should be a disciplined approach towards recording historical data to make them reliable and readily available. Automated tools and standards for recording data would greatly help in this regard.

3 Future Direction

The empirical study we conducted here is an early start for understanding the subleties of NOV parameters that we have discussed in [5]. One of our research goals, as an extension to the work we have described in this paper, is to formulate indirect measurements of σ in terms of the environment parameters that are specific to the particular examples we choose to collect historical information. Generalizing these results would be impossible unless a broad spectrum of examples is covered.

We list the following set of tasks as the fundamental problems to be solved in this regard:

- Developing a representational model for measuring σ based on: (i) its mathematical significance in the NOV framework, and (ii) the general understanding and known heuristics about it.

- Developing a measurement function for σ and figuring out the historical data/metrics to collect based on the model.

- Verifying our measurement function for σ based on the data from real world projects.

Formulating *the* standard model for σ is a challenging task. Our results in this study have provided an early positive hint that more dependencies on external parameters suggests higher technical potential of a module. This is a suggestive validation of the current understanding of σ within the scope of examples as used in [11].

However a lot of work remains to be done to formulate a correct model for σ considering the fundamental principles of software measurement and well established prerequisites for validating such measurement. [8]. At this point very little is known about the nature of σ, for example, the kind of relationships that exist between modules based on the value of σ cannot be established universally. We believe, building on our current understanding about σ as outlined in this paper, we will be able to develop the required prerequisites for validating current heuristics and understanding about σ.

4. Conclusion

We have presented an empirical study to validate a general understanding about an important parameter of NOV, a quantitaive model to evaluate design options, called the technical potential of a module. We collected historical data from CVS and issues database of ArgoUML, an open source project, to show that the increase in a module's dependency to External Parameters increases its technical potential.

Our results support the assumptions made about σ in the early works in using NOV. This suggests that scaling our study to several well documented systems would further unveil interesting properties and relationships that hold between the software attributes and parameters in NOV. These large scale studies are needed to establish NOV as a reliable decision making tool in evaluating software design options. As an analytical framework, we find NOV very promising as it ties together modular dependencies, uncertainty, and economic theory in a cohesive model. We foresee further research effort in this field would add significant contributions in the field of value based design and software engineering at large.

References

[1] Argouml, project announcement. *http://argouml.tigris.org/servlets/NewsItemView?newsItemID=1019*.

[2] Dependency finder home page. *http://depfind.sourceforge.net*.

[3] A. Ant. Ant build system home page. *http://ant.apache.org*.

[4] ArgoUML. Argouml home page. *http://argouml.tigris.org*.

[5] S. K. Bajracharya, T. C. Ngo, and C. V. Lopes. On using net options value as a value based design framework. In *The 7th International Workshop on Economics-Driven Software Engineering Research*. To Appear.

[6] C. Y. Baldwin and K. B. Clark. *Design Rules vol I, The Power of Modularity*. MIT Press, 2000.

[7] de Souza Cleidson, P. Dourish, D. Redmiles, S. Quirk, and E. Trainer. From technical dependencies to social dependencies. In *CSCW'04 Workshop on Social Networks, Chicago, IL, USA*, November 6-10 2004.

[8] N. Fenton. *Software metrics (1st ed.): A rigorous approach*. Chapman and Hall, 1991.

[9] D. M. German and A. Mockus. Automating the measurement of open source projects. In *Proceedings of the 3rd Workshop on Open Source Software Engineering*, May 2003.

[10] C. V. Lopes. On the nature of aspects: Principles of aspect-oriented design. In *ACM Transactions of Software Engineering*. Under Review.

[11] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 15–26, New York, NY, USA, 2005. ACM Press.

[12] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[13] J. E. Robbins, D. M. Hilbert, and D. F. Redmiles. Extending design environments to software architecture design. In *The International Journal of Automated Software Engineering. Special issue: The Best of KBSE'96*, 1996.

[14] G. Succi, J. Paulson, and A. Eberlien. Preliminary results from an experimental study on the growth of open source and commercial software products. In *Third International Workshop on Economics-Driven Software Engineering Research (EDSER 03), Toronto, Canada*, May 2001.

[15] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 99–108. ACM Press, 2001.

[16] W3C. Xml path language (xpath). *http://www.w3.org/TR/xpath*.