View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by AIS Electronic Library (AISeL)

JITTA

JOURNAL OF INFORMATION TECHNOLOGY THEORY AND APPLICATION

DOES THE "GOLDILOCKS CONJECTURE" APPLY TO SOFTWARE REUSE?

DEREK L. NAZARETH, University of Wisconsin - Milwaukee
Sheldon B. Lubar School of Business, Milwaukee, WI 53201, Tel: 414-229-6822, Fax: 414-229-5999, Email: derek@uwm.edu

MARCUS A. ROTHENBERGER, University of Nevada Las Vegas
Department of MIS, College of Business, Las Vegas, NV 89154-6034, Tel: 702-895-2890, Fax: 702-895-0802, Email: marcus.rothenberger@unlv.edu

ABSTRACT

Adopters of corporate software reuse programs face important decisions with respect to the size of components added to the reuse repository. Large components offer substantial savings when reused but limited opportunity for reuse; small components afford greater opportunity for reuse, but with less payoff. This suggests the possibility of an "optimal" component size, where the reuse benefit is at a maximum. In the software engineering discipline, this relationship – termed the Goldilocks Principle - has been empirically observed in software development, software testing, and software maintenance. This paper examines whether this relationship also applies for software reuse. In order to understand the effects of component size and repository size on the benefits of a reuse program this paper extends an empirically grounded reuse model to assess the effects of component size on reuse savings. The study finds that a variant of the Goldilocks Principle applies with respect to both component and repository size, suggesting that uncontrolled growth of a reuse repository and an inappropriate choice of component size may reduce benefits obtained from reuse.

INTRODUCTION

Software development is generally acknowledged as an expensive and lengthy process, often producing artifacts that are of suspect quality and maintainability. Sustained growth in the demand for software, coupled with shortages in the supply of software developers and the stagnant productivity in software development, has exacerbated the problem. Several different strategies have been proposed to alleviate this, including software automation, outsourcing, use of agile methodologies, and software reuse, among others. Each of these approaches provides some relief, at the expense of related objectives. This paper focuses on software reuse as a possible strategy for alleviating the software development crunch. The benefits

Ken Peffers acted as the senior editor for this paper.

Nazareth, D. L., and M. A. Rothenberger, "Does the 'Goldilocks Conjecture' Apply to Software Reuse?" *Journal of Information Technology Theory and Application (JITTA),* 8:2, 2006, 57-67.

claimed for software reuse include reduced development cost and time, improved software quality, increased developer productivity, and improved software maintainability (Ravichandran and Rothenberger 2004). These benefits are offset by the cost of setting up a repository of reusable components, and ongoing population and management of the repository. It is expected that long term savings through software reuse will outweigh the initial costs of adopting a reuse program (Lim 1998). Most studies addressing costs and benefits of software reuse tend to focus on post-hoc analyses of cost data for a portfolio of projects. Frakes and Kang (2005) identify a need for more measurement and experimentation of reuse. This paper uses an empirically grounded model of reuse to address this need, examining how component size and repository size determine viability of a software reuse program. This approach allows us to analyze the effect that changes to repository and component size have on the economic feasibility of a reuse program. Intuitively, it would appear that very small components may be usable in a large number of applications; however, each reuse instance will provide little savings over traditional development. In such a scenario, the search and retrieval costs that are expended to find the component in a repository may offset a substantial portion of the savings obtained through reuse. Thus, the reuse program may not provide net positive benefits. On the other hand, very large components are more specific and are therefore expected to be reusable in fewer applications; however, each reuse instance will provide larger savings over traditional development. Search and retrieval costs are expected to be small in comparison to the savings obtained by reusing large components. Consequently, the reuse program may appear to pay for itself. However, as the repository size grows, management, maintenance, and search and retrieval costs also grow, offsetting some of the reuse savings, thereby leading to an uneconomical reuse program. Our model investigates whether reuse programs may be subject to optimality considerations, because of these tradeoffs, wherein performance on some pre-specified dimension initially improves, but later degrades.

CONTRIBUTIONS

Most software reuse research focuses on the analysis of empirical data and reporting of case evidence. While this has provided important insights about reuse success factors, reuse methods, and reuse benefits, there has been little research done that examines the complex interdependencies of reuse program-related decisions. This research addresses this need by examining whether repository size and component size affect the benefits of reuse, and whether there are optimal levels for both.

The contributions of this study are twofold. First, it provides evidence that uncontrolled repository growth leads to a reduction of reuse benefit as the search cost in a larger repository outweighs the benefits obtained from the increase in reuse opportunities. While it is generally agreed on that a very small repository cannot lead to substantial reuse, the notion of optimality in repository size is novel. Second, the study indicates that there is also a preferred component size. Very large and very small components reduce the reuse benefit. Very small components provide not enough reuse benefit per reuse instance to offset the search and retrieval cost; very large components reduce reuse opportunities, as their requirements become too specific.

THE GOLDILOCKS CONJECTURE

The concept of optimality has fascinated researchers in the software engineering discipline. Optimality introduces the notion of the best possible performance on a given dimension. The application of curve fitting techniques, particularly non-linear models, to empirical software engineering data suggests the presence of an optimal value. Early studies in the area of software development have established a non-linear relationship between module size and development effort that suggests an optimal module size (Bowen 1984). A host of explanations have been offered for this phenomenon. These include tradeoffs between the complexity of the interface between modules, and the inherent complexity of the

code. For smaller modules, the interface complexity contributes disproportionately, while larger modules are more likely to be influenced by code complexity.

In the software quality discipline, several studies have examined the relationship between software defects and module size. Empirical data has suggested the presence of non-linear relationships, and polynomial curve fitting techniques have been employed in the creation of defect prediction models (Compton and Withrow 1990; Gaffney 1984). An unintended consequence of using models with polynomial terms is that the model suggests the presence of an optimal size of the component in the context of defect reduction. Gaffney (1984) examined defect densities for assembly language components, and the resulting model predicted the density to be lowest for components of size 877 lines. In a separate exercise on Ada components, using a different polynomial model, Compton and Withrow (1990) concluded that the component size that yielded the lowest defect densities was 83 lines. They dubbed this the "Goldilocks Principle", based on the notion that there exists an optimal component size that is "not too big nor too small". Other researchers have also experienced similar results when analyzing code defect densities. Several possible explanations have been advanced in this context, including disproportionate user interface defect that skew the densities for smaller components, more attention to the development of larger components, and human cognitive processing limitations that cause the introduction of more defects for larger components.

These results have some definite implications for software development. First, they suggest that component size is a determinant of defect density. This is at odds with the notion of software decomposition, which seeks to break up components into smaller, more easily crafted, and potentially more reusable components. Further, it provides a pessimistic outlook for developers engaged in the creation of very small or very large components. An excellent critique is provided by Fenton and Neil (1999), where they conclude that "the relationship between defects and component size is too complex, in general, to admit to straightforward curve fitting models". Their analysis and results would appear to contradict the notion of the "Goldilocks Conjecture" as an underlying relationship between defect density and component size. Despite the concerns raised in (Fenton and Neil 1999), there is evidence to suggest that some data sets support the conjecture. It should be borne in mind that the data, however problematic from a quality perspective, merely represents some underlying facts. As such, it cautions developers of very small and very large components about the propensity for higher defect densities.

A similar relation is also observed in software maintenance, wherein maintenance effort is high for small modules, then decreases as the module size increases, and finally increases once again for large modules (Banker et al. 1993a). This research used a model to predict maintenance effort as a function of procedure size and other cost drivers, and obtained a U-shaped relationship, with an "optimal" procedure size of 44 executable lines of code. As with other non-linear relationships, caution should be exercised in designating the low-cost values as the optimal size. Clearly, the component size will be dictated by functional requirements. However, the implications for software maintenance are undeniable.

The presence of non-linear relationships between cost and size in the development, quality assurance, and maintenance of software is intriguing. The rest of this paper seeks to examine whether such relationships also exist for software reuse.

THE GOLDILOCKS CONJECTURE AND SOFTWARE REUSE

There is little doubt that software reuse can generate savings in development effort. However, setting up a repository of reusable components and searching it for appropriate modules in an reuse context entails definite costs. It is expected that the initial phases of software reuse in an organization will be characterized by higher setup costs vis-à-vis savings from reuse. As the repository grows larger, it is expected that the savings through reuse will start to offset the costs associated

with the initial setup, and a breakeven point will be reached. Beyond this, the savings should continue to outpace costs, as cataloging costs and search costs are expected to be smaller than development costs averted through reuse. This paper focuses on the breakeven point for software reuse. Of particular interest is whether the breakeven point occurs differently if the repository is populated with small components, or large components. If the breakeven varies with component size, the Goldilocks Conjecture may apply.

A review of the software reuse literature did not yield any insight into the effect of component size on the extent of software reuse. However, data from software reuse studies have indicated several non-linear relationships between various reuse parameters and overall reuse costs. In a study of 2954 reused components at NASA, Selby (1988) determined that a concave non-linear relationship existed between modification effort and percentage of code modified as part of the reuse effort, whereby small modifications generated disproportionately large costs. Gerlich and Denskat (1994) posit that changes to multiple components in an application will generate a non-linear set of changes to their interfaces. Cost estimation models for software development in the presence of reuse and reengineering also include non-linear drivers (Clark et al. 1998). Non-linearities involving a second order relationship between component size and other software reuse parameters would indicate support for the Goldilocks Conjecture.

REUSE BREAKEVEN AND COMPONENT SIZE

To investigate a possible relationship between the breakeven point for software reuse and component size, this research employs a domain-specific model of systematic software reuse. The motivation for constraining the model to work with a single domain stems from the notion that reuse is expected to be greatest when the repository of reusable components address a set of related applications from the same domain. Reuse across domains is expected to be limited, and presents a less interesting scenario. The model addresses cost factors and savings relating to systematic software reuse. An earlier version of the model that was directed at capturing the effect of project size on savings is described in (Nazareth and Rothenberger 2004). The model has been further enhanced to permit investigating the relationship between component size and repository size. A summary of the model is presented in Table 1. A more detailed discussion of the analysis of the underlying relationships established in the model is presented in (Nazareth and Rothenberger 2004). The model is highly parameterized and employs multiple mechanisms to accommodate several different reuse strategies. The model has undergone substantial testing, using a wide range of coefficient values. The lack of brittleness in its behavior suggests that it is a robust model. The prior study established a proportionate relationship between project size and reuse savings, indicating that savings tended to increase uniformly with project size, other conditions being the same. It can therefore be inferred that project size is not likely to be an issue in this analysis. On the other hand, we have learned that the repository size affects the search cost, as well as the likelihood to find desired components. Further, the component size also affects the leverage of each reuse instance. These observations suggest that both will have an impact on the breakeven point.

The model was calibrated using empirical data from reuse projects, in conjunction with other findings from the reuse literature. Results from the model will clearly be shaped by this calibration. The model can easily be recalibrated for other settings, which would yield different numbers, but similar trends due to its robust nature. As with any model-driven analysis, any findings should be viewed in light of the overall trends, rather than absolute values. Calibrating the model required that appropriate values for the relative effectiveness coefficients for query formulation, retrieval, modification, making a component generic, and cataloging needed to be determined, vis-à-vis development effort. This study employed an overall development effectiveness coefficient of 10 lines-of-code per time unit which allowed us to derive the other effectiveness coefficients separately. Cataloging cost are expected to be low compared to development cost, query and

Table 1. The Reuse Model [adapted from (Nazareth and Rothenberger 2004)]

The savings of developing a project with reuse over developing from the ground up is based on the total cost assuming no reuse ($C_{noreuse}$) and the total cost of the project with reuse (C_{reuse}):

$C_{savings} = C_{noreuse} - C_{reuse}$

The total cost of a project assuming no reuse is assessed based on the development cost (C_D) and the number of components in the project (P_C):

$C_{noreuse} = C_D \times P_C$, where

- The development cost is based on component size (S), a factor that addresses development economies of scale (β), and the development effectiveness of the developers (D_E): $C_D = S^{\beta} \times D_E$

The total cost of a project developed with reuse is based on the cost of components developed from scratch (C_{dev}), the search cost expended (C_{sea}), the cost of publishing new components in the repository (C_{rep}), and the cost of modifying existing components for the current project (C_{mod}):

$C_{reuse} = C_{dev} + C_{sea} + C_{mod} + C_{rep}$

The total cost of custom development on a project is based on the component Size (S), a factor that addresses development economies of scale (*β*), the number of components newly created for the project (N_N) and the development effectiveness of the developers (D_E):

$C_{dev} = S^{\beta} \times D_E \times N_N$

The search costs (C_{sea}) address efforts required to locate appropriate components for reuse in a new software development project. They include query formulation costs (C_Q), retrieval costs (C_R), and are based on the number of components in the project (P_C):

$C_{sea} = (C_Q + C_R) \times P_C$, where

- Query formulation costs are based on the number of terms to be retrieved (n_q) moderated by the developers' effectiveness of selecting among the query criteria (Q_E): $C_Q = n_q \times Q_E$.
- Retrieval costs are based on the number of components in the repository (N), the number of query criteria (n_q), the selectivity among criteria (a), and the developers' effectiveness of retrieval (R_E):

$$C_R = N \times a^{n_q} \times \frac{1}{1 + .2 \times (n_q - 1)} \times R_E$$

The cost of modifying an existing component for a project (C_{mod}) is based on the modification cost (C_M), the cataloging cost (C_C), and the number of components reused with modification (N_M):

$C_{mod} = (C_M + C_C) \times N_M$, where

- Modification costs are assessed based on the degree of fit of the retrieved components (s), on the effectiveness of the developers to modify a component (M_E), its size (*S*), and an economies of scale factor (*β*): $C_M = (1-s) \times S^{\beta} \times M_E$
- Cataloging costs are modeled on the basis of the number of cataloging dimensions (n_c) and the effectiveness of the developers to catalog a component (C_E): $C_C = n_c \times C_E$

The cost of publishing components as part of an actual reuse project (C_{rep}) represent the effort associated with the addition of new components to the repository. This includes the cost of cataloging the components (C_C), the cost of making a component more general to improve reusability (C_G), and the number of components to be published (N_P):

$C_{rep} = (C_G + C_C) \times N_P$, where

- The cost of making a component generic is based on the component size (*S*), an economies of scale factor (*β*), and the developers' effectiveness of making a component generic (G_E): $C_G = S^{\beta} \times G_E$

The number of components to be published (N_P), the number of components to be written from scratch (N_N), and the number of components to be modified (N_M) depend directly on the proportions of components reused (*r*), reused as is (*p*), and the degree of fit of retrieved components (*s*); these values will increase as the size of the repository grows. The slopes of these three variables have been carefully calibrated. Many of above relationships incorporate relative effectiveness coefficients (Q_E, R_E, M_E, G_E, and C_E), which were calibrated based on prior research findings, in order to ensure realistic results for our analysis.

retrieval cost are expected to be low compared to modification and cataloging cost, thus the respective effectiveness coefficients are set to yield small values in comparison. Modification effectiveness is closely related to development effectiveness; however, since modification requires the understanding of code developed by other programmers, modification effectiveness is expected to be slightly lower. Finally, the effectiveness to make components generic is expressed as a fraction of development effectiveness. We selected the median of the data of several projects according to Poulin (1997), which suggests that developing a component for reuse requires 50% additional development effort than developing the same component for only one project. Nazareth and Rothenberger (2004) discuss additional details on this part of the model's calibration.

When searching for behavior consistent with the Goldilocks Principle, a large space may need to be explored. However, the phenomenon is likely to be localized, and values for repository size and the average component size need to be attuned. The conditions employed in this study are as follows: The repository size was varied from 0 to 2000 components in steps of 20. The average component size was varied from 20 to 200 lines of code in steps of 1. Expected savings from reuse are computed for these conditions, with particular emphasis on when the breakeven occurs. The analysis was performed for cases of decreasing, constant, and increasing economies of scale for software development. The results are depicted in Figure 1, and represent the case for increasing economies of scale. Similar results were obtained for the other cases. For very small repositories no breakeven was attained, indicating repository creation costs outweighed any savings through reuse. The analysis was repeated for different search costs, both lower and higher than that depicted in the figure. Similar trends were observed, with a flatter curve for lower search costs, and a steeper curve for larger search costs. The results suggest that there is a clear minimum component size needed for the reuse program to break even.

REUSE SAVINGS AND COMPONENT SIZE

To determine whether there is an "optimal "component size, the average component size was varied from 10 to 1000 lines of code for a fixed repository size. Expected savings from reuse are computed for these conditions. A repository of 800 components allows for a reuse program to break-even with relatively small components (see Figure 1), representing an intriguing case that merits further investigation. Figure 2 depicts the overall savings that can be expected for a repository of 800 components, as the average component size is varied. For small components the savings are negative, as search and retrieval costs outweighed the benefit of reusing these small components. However, as the components grow larger, the savings through reuse outpace the search and repository maintenance costs. For very large components, a reversal is observed due to the low potential for reuse of these complex components. The results are displayed on a log scale to better illustrate the non-linearity. Peak values of savings are obtained for component sizes of 240 lines of code, which is considerably larger than that advocated by most proponents of modular software development. Similar trends were observed when the analysis was repeated for different repository sizes. Figure 3 shows the reuse savings for a considerably larger repository comprising 1,600 components. In this case, the peak value was observed at 290 lines of code.

IMPLICATIONS

While it is tempting to focus solely on the Goldilocks Conjecture and examine overall reuse savings in light of changing component size, we believe that the breakeven analysis offered in Figure 1 has equally important implications for reuse program managers. As the repository size changes, the average size of the component in the repository required to break even also changes. Our findings suggest that a variant of the Goldilocks Principle applies for the breakeven analysis of software reuse programs. The unstated implication is that a minimum component size is needed for reuse programs to be economical, below which

no reuse program is ever economical. For very small repositories, breakeven simply does not occur, consistent with the notion that the initial costs to set up the repository are not likely to be offset by the limited opportunities for reuse. As the repository grows, a more moderate component size is needed to break even. Intuitively, this is logical in that small components are not likely to generate

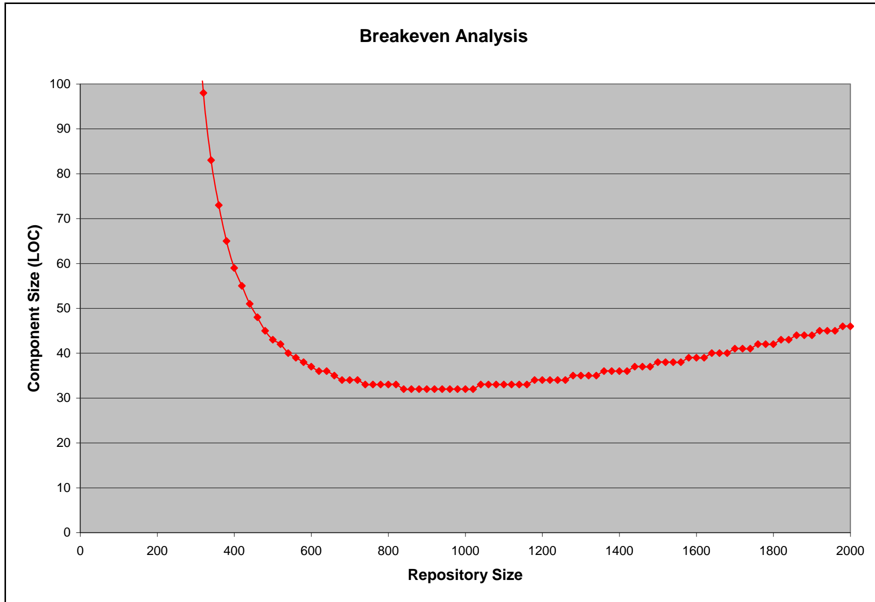

Figure 1: Breakeven Component Size for Software Reuse

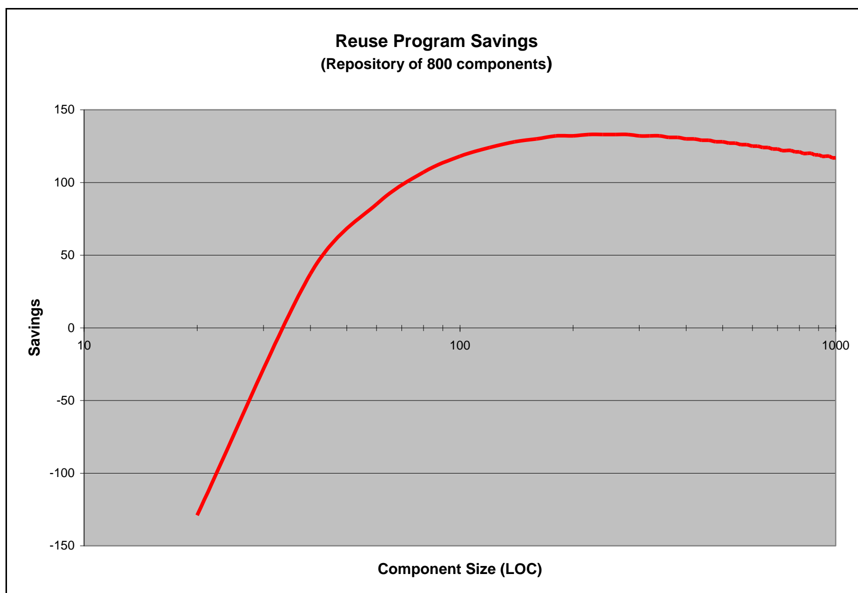

Figure 2: Software Reuse Savings as a Function of Component Size (for a Repository of 800 Components)

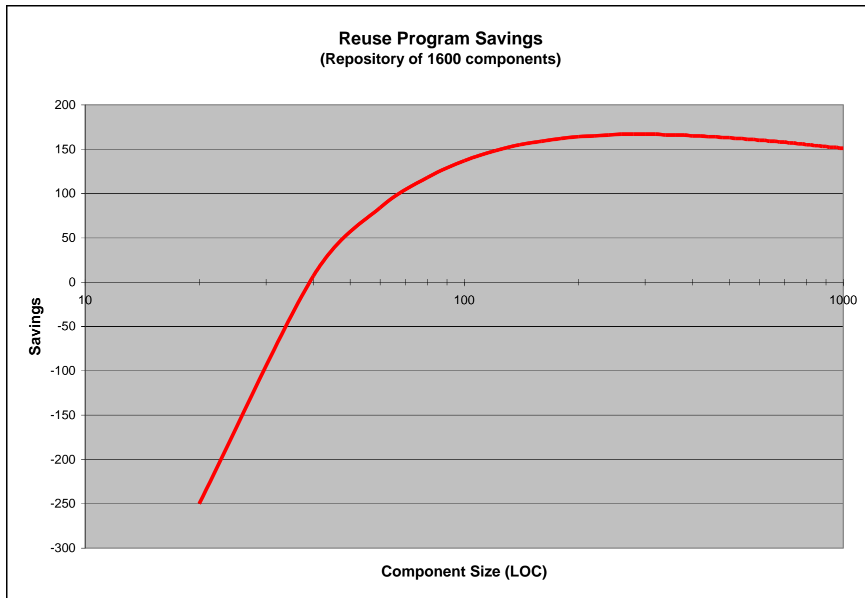

Figure 3: Software Reuse Savings as a Function of Component Size (for a Repository of 1,600 Components)

sufficient savings when reused. With a larger repository, there are more opportunities for reuse, and hence less need to rely on large components for savings through reuse. However, as the repository size increases even more, the average size of components required to break even begins to climb. This can be explained in the light of increased search, maintenance, and modification costs associated with large repositories. These increased costs need to be offset through increased savings, and hence larger components to be reused.

Figures 2 and 3 focus on the more traditional implications of the Goldilocks conjecture, i.e. there is an "optimal" component size, at which the performance on a predetermined metric is best. These figures examine the effect of different component sizes on savings, assuming a constant repository size. The graphs suggest that not all components are likely to generate similar savings through reuse. While it is intuitive that small components do not provide sufficient leverage to outweigh the cost of their search and retrieval, the dip in savings for very large components necessitates an alternative explanation: the reduction of reuse opportunities of very large and specialized components may explain this phenomenon. Thus, it appears that declining reuse opportunities of larger components can not always be offset by an increase in the number of components to choose from, as the search cost for large repositories tends to overwhelm the savings through limited reuse. However, the savings tail off only for large components sizes (about 240 lines of code).

These results are examined in the light of earlier findings in software reuse, as well as other areas of software engineering. Most empirical studies in software reuse tend to work with a limited number of reuse programs, affording little opportunity to generalize findings. Consistent with the findings from our model, anecdotal evidence suggests that larger repositories will lead to more reuse and thus increased savings from reuse (Banker et al., 1993b). Also, anecdotal evidence reports that a larger component size leads to an increase in reuse payoffs (Apte et al. 1990). However, any assessment as to whether the savings will increase, remain roughly constant, or decrease, is not possible in empirical studies, based on lack of comparability of the data. The use of a robust model, that is rigorously constructed and calibrated, permit systematic exploration of the benefits of a reuse program. The results indicate that the Goldilocks Conjecture also applies to software reuse. This is not inconsistent with the

findings in software development, software quality assurance, and software maintenance.

These results have clear implications for software reuse. While they confirm that extremely small repositories are not likely to generate any meaningful savings, they also indicate that extremely small components may not generate enough savings to be economically worthwhile. Likewise, caution needs to be exercised with respect to large repositories. Table 2 summarizes the implications of this study. Very small components and very small repositories inhibit a reuse program from breaking even. While this may appear to cover the bulk of the table, the implications are not quite as bleak. Breakeven occurs when the repository contain about 200 to 250 components. Most reuse programs will experience negative savings while the repository is initially assembled. However, attempting reuse on a low-level of abstraction does not generate a net saving over traditional development, as the benefits of reusing very small components do not offset the effort invested in cataloging, searching, and retrieving the components. For the conditions explored, the model suggests that the minimum component size needed to break even is approximately 30 lines of code. The center cell in the table represents the conditions when the reuse program is expected to be most effective. Moderately sized components (between 30 and 240 lines of code), in moderately sized repositories (between 400 and 1000 components) generate considerable savings. As the component size grows, the savings will dip, but the overall reuse program still remains economical. Likewise, as the repository size grows, it takes a larger component to break even, but this is still characterized by overall savings through reuse. Repositories are expected to grow over time, as components developed for ongoing projects are added to the reuse library. Under these circumstances, search and retrieval cost per reuse instance increase. Reuse managers need to be aware that this necessitates larger components to break even. While the average size may not fall below the breakeven point, the overall savings through reuse will be lower. Very large components in moderate repositories generate reuse savings; however, the benefits from reuse are reduced, on account of fewer opportunities for reuse. Very large components are highly specific and the probability of finding a suitable match in a new project is small. Nevertheless, even under these conditions the reuse program breaks even, because each reuse instance represents a significant saving over developing an equally large component, allowing the reuse program to sustain itself with fewer reuse opportunities. In a similar vein, large components in a large repository will still generate some savings – just not as much as the center cell scenario, where search and maintenance costs are lower, and opportunities for reuse are greater. The lack of reuse opportunities of individual large components in this case is offset by the collective increase in reuse opportunities from the large repository.

As with all analysis involving the Goldilocks Conjecture, it should be stressed that the goal is not to determine an optimum around which the reuse program should be structured. Rather, it provides a basis for identifying implications for operating in conditions that stray from these preferred areas. In particular, the paper makes a case for moderate component size and for controlled growth of the repository.

CONCLUSION AND LIMITATIONS

This study employed a domain-specific model to provide greater insight into the economics of a software reuse program. Two forms of analysis were performed – a breakeven analysis to assess whether reuse is worthwhile, and a more traditional cost-benefit analysis, which suggests that savings from reuse may eventually tail off as reuse components grow larger. The findings suggest that the Goldilocks Conjecture does apply, both for breakeven as well as for reuse savings assessments. For the breakeven analysis, the quadratic relationship is observed between the component size required to break even and different repository sizes. This suggests that software reuse programs can never break even if they are populated with very small components, no matter how many components are available to reuse. The second analysis also found a quadratic relationship between overall reuse savings and component size, representing the classic interpretation of the Goldilocks Conjecture. The analysis was

Table 2. Implications for Reuse Program Effectiveness

Component Size		Repository Size: Very Small	Repository Size: Moderate	Repository Size: Very Large
Component Size	Very Small	Reuse program not economical	Reuse program not economical	Reuse program not economical
Component Size	Moderate	Reuse program not economical	Reuse program preferred to traditional software development	Reuse program viable but savings may dip due to higher repository management and search costs
Component Size	Very Large	Reuse program not economical	Reuse program viable but savings may dip due to fewer reuse opportunities	Reuse program viable but savings may dip due to fewer reuse opportunities and higher repository management and search costs

repeated for multiple repository sizes, with similar results, once again indicating that the savings will eventually tail off, though the peak is observed at slightly different points.

It is not the intent of this study to prescribe specific numbers for repository and component size – these will be very much context dependent. Instead, it identifies conditions where a reuse program is economically viable. These results should caution reuse managers about small repositories, small reusable components, and uncontrolled repository growth.

The model employed in this study is deterministic in nature, assuming average component size instead of a portfolio of components of various sizes. Moreover, it assumes that the components all pertain to a specific business domain. Clearly, these are restrictive assumptions. Projects are expected to comprise components of various sizes, and as a consequence, the repository is also expected to include components of varying size. Likewise, not all applications would pertain to the same business domain, thereby reducing some opportunities for reuse. Future research into this phenomenon will involve the need for a more comprehensive simulation model in which projects are assembled from different business domains and involve a portfolio of components with varying size and propensity for reuse. This approach would also permit dynamic growth of the repository, with initial projects contributing heavily to the repository and subsequent projects contributing only the unique processing that is not covered elsewhere among existing applications.

REFERENCES

Apte, U., C.S. Sankar, M. Thakur, and J.E. Turner, "Reusability-Based Strategy for Development of Information Systems: Implementation Experience of a Bank," *MIS Quarterly*, 1990, 14:4, pp 420-433.

Banker, R.D., S.M. Datar, C.F. Kemerer, and D. Zweig, "Software Complexity and Maintenance Costs," *Communications of the ACM,* 1993a, 36:11, pp. 81-94.

Banker, R.D., R.J. Kauffman, and D. Zweig, "Repository Evaluation of Software Reuse," *IEEE Transactions on Software Engineering*, 1993b, 19:4, pp 379-389.

Bowen, J.B., "Module size: A Standard or Heuristic," *Journal of Systems and Software*, 1984, 4, pp. 327-332.

Clark, B., S. Devnani-Chulani, and B. Boehm, "Calibrating the COCOMO II Post-Architecture Model," *Proceedings of the 20th International Conference on Software Engineering ICSE-20*, Kyoto, Japan, 1998, pp. 477-480.

Compton, B.T., and C. Withrow, "Prediction and Control of Ada Software Defects," *Journal of Systems and Software*, 1990, 12:3, pp. 199-207.

Fenton, N.E., and M. Neil, "A Critique of Software Defect Prediction Models," *IEEE Transactions on Software Engineering*, 1999, 25:6, pp. 675-689.

Frakes, W.B. and K. Kang, "Software Reuse Research: Status and Future," *IEEE Transactions on Software Engineering*, 2005, 31:7, pp. 529- 536.

Gaffney, J.R., "Estimating the Number of Faults in Code," *IEEE Transactions on Software Engineering*, 1984, 10:4, pp. 459-464.

Gerlich, R., and U. Denskat, "A Cost Estimation Model for Maintenance and High Reuse," *Proceedings of the European Software Cost Modeling Conference*, Ivrea, Italy, 1994.

Lim, W.C., *Managing Software Reuse: A Comprehensive Guide to Strategically Reengineering the Organization for Reusable Components*, Prentice-Hall, Upper Saddle River, NJ, 1998.

Nazareth, D.L. and M.A. Rothenberger, "Assessing the Cost-Effectiveness of Software Reuse: A Model for Systematic Reuse," *Journal of Systems and Software*, 2004, 73:2, pp. 245-255.

Ravichandran, T. and M.A. Rothenberger, "Black Box or White Box Reuse: The Impact of Component Markets on Reuse Strategies," *Communications of the ACM*, 2003, 46:7, pp. 109-114.

Selby, R., "Empirically Analyzing Software Reuse in a Production Environment," in *Software Reuse: Emerging Technology*, W. Tracz (Ed.), IEEE Computer Society Press, 1988, pp. 176-189.

AUTHORS

Derek L. Nazareth is Associate Professor of Management Information Systems at the University of Wisconsin-Milwaukee. He holds a Ph.D. in Management from Case Western Reserve University. His current research interests include application development using web services, software reuse, and information systems ethics. His papers appear in *Communications of the ACM, IEEE Transactions on Knowledge and Data Engineering, Journal of Management Information Systems, Decision Support Systems, Knowledge Acquisition, OMEGA, Information & Management* among others. He is a member of AIS, ACM, and INFORMS, and was the Program Chair for AMCIS 1999, and the Treasurer for ICIS 2006.

Marcus Rothenberger is an Associate Professor in the Department of Management Information Systems. He holds a Ph.D. in Information Systems from Arizona State University. Dr. Rothenberger's current research explores software process improvement issues, software reusability, performance measurement, software development offshoring, and the adoption of enterprise resource planning systems. His work has appeared in major academic journals, such as the *Decision Sciences Journal, IEEE Transactions on Software Engineering*, *Communications of the ACM*, and *Information & Management*. Dr. Rothenberger is a senior editor of the *Journal of Information Technology Theory and Application*. He has been actively involved in editorial and program committee capacities in several academic conferences, including the *International Conference on Information Systems* and the *Americas Conference on Information Systems*.