# The Potential of Portfolio Analysis in Guiding Software Decisions

S. Butler* P. Chalasani† S. Jha* O. Raz* M. Shaw*

March 16, 1999

## Abstract

Developing a complex software system involves decisions about how to allocate a limited resource budget among a collection of costly software alternatives (such as technologies or analysis techniques) that have uncertain future benefits. Very little quantitative guidance is currently available to make these decisions. We suggest that these allocation problems are naturally viewed in the powerful portfolio selection framework of financial investment theory. We view each software activity as an investment opportunity (or security), the benefit from the activity as the return on investment, and the allocation problem as one of selecting the "optimal" portfolio of securities.

## 1 Introduction

With the rapidly growing complexity of software systems, developers are increasingly facing the problem of allocating their limited resources (such as programmer time and wages, disk space, CPU time, off-the-shelf software costs, etc.) among several activities that contribute to a given goal (such as testing/prototyping, verification, code restructuring, analysis, validation), or among several overlapping technologies (such as passwords, encryption, authorization lists, physical isolation, firewalls, and activity monitors for security). To complicate matters, these activities and technologies typically have *uncertain* future benefits.

*School of Computer Science, Carnegie Mellon University, Pittsburgh, PA

†Department of Computer Science, Arizona State University, Tempe, AZ

Currently, decisions such as these are made on an ad-hoc basis, using feature comparison, subjective judgement, gut feel, persuasiveness of the salesman, or tradition. The software developer may make a good-faith effort to compare the alternatives, or even to evaluate the future consequences of the alternatives. However there is very little systematic guidance available to software engineers.

Therefore, there is a critical need for rigorous quantitative models of resource allocation problems for software systems. In this paper we suggest that it is natural to view these problems as *portfolio selection* problems. That is, the various activities are viewed as investment opportunities, and the developer systematically decides what portion of his resource pool, if any, to invest in each opportunity, in such a way that a certain globally "best" (in some appropriate sense) allocation is achieved. This is analogous to an investor deciding how much of his budget to invest in each of a collection of stocks and other securities. The return on investment for each security is a random variable (with an assumed distribution), just like the benefit from investing in a software activity.

It may seem as if one could just select the security (or activity) that has the best expected future return, and invest all the available budget in that security. However this strategy ignores the important notion of *risk*, i.e., the variance of the return on investment in the stock. Investors may prefer to get smaller expected returns in exchange for lower variability of the return, i.e., lower risk. Portfolio analysis quantifies how investors can optimally make this tradeoff between risk and expected return.

We envisage three possible benefits of viewing the above types of software decision problems as a portfolio selection problem: (a) It can be used to actually guide the decisions in a specific software project, (b) it can provide rough general design principles and qualitative rules of thumb for such decisions, and (c) it can be used to rationalize (or perhaps even refute) guidelines that have been accepted in the software community.

In this paper we elaborate on conventional portfolio analysis; discuss the correspondence of the portfolio selection problem to the software decision problem; describe specific applications to the problems of choosing security technology and software validation techniques, and assess the prospects of this approach.

## 2 The Portfolio selection problem

In this section we briefly summarize the single period portfolio optimization problem. For more details please see [JI87]. Assume that we have $n$ assets $A_1, \cdots, A_n$. Let $z_i$ be the return per dollar on the $i$-th investment $A_i$ . Since the return $z_i$ is not assumed to be known a priori, it is a random variable. In general, $z_i$ depends on future events and is random because the future is uncertain. The *returns vectors* $\vec{z}$ is $(z_1, \cdots, z_n)$, i.e., the vector of returns for each investment. Suppose an investor has $B$ dollars to invest. Let $\pi_i$ be the fraction of the wealth invested in the $i$-th asset $A_i$. An investment strategy can be viewed as a vector $\vec{\pi} = [\pi_1, \cdots, \pi_n]^T$, which is called a *portfolio*. We also have:

$$\pi_i \geq 0 \ , \quad \sum_{i=1}^{n} \pi_i \ = \ 1$$

Notice that $\pi_i \geq 0$ means that we do not allow *short selling* of assets, i.e., selling assets as opposed to buying them. Short selling is allowed in security markets, but does not seem applicable in our context. The portfolio $\vec{\pi}$ leads to the following random cash flow:

$$\begin{aligned} Z_{\vec{\pi}} &= B\vec{\pi}^T\vec{z} \\ &= B\sum_{i=1}^{n} \pi_i z_i \end{aligned}$$

We assume that the investor in question has an utility function $u$. We assume that $u$ is strictly increasing and concave (the justification for this can be found in utility theory [Fis70]). Utility function models the preference of an investor. If we have two random cash flows $z_1$ and $z_2$ and $E[u(z_1)] \geq E[u(z_2)]$, then the investor prefers $z_1$ over $z_2$. In other words, the investor prefers the cash flow with higher expected utility. Many kind of investor preferences can be captured by utility functions (please see [Fis70]).

Given the *returns vector* $\vec{z}$, initial wealth $B$, and an utility function $u$, the investor wants to find a portfolio $\vec{\pi}$ that maximizes the expected utility of the random cash-flow $E[u(B\vec{\pi}^T\vec{z})] = E[u(BZ_{\vec{\pi}})]$ generated by the investment. (A vector with superscript $T$ denotes its transpose). Formally, an investor wants to solve the following constrained global optimization problem:

$$\begin{gathered} \max_{\vec{\pi}} E[u(B\vec{\pi}^T\vec{z})] \\ \pi_i \geq 0 \ , \quad \textstyle\sum_{i=1}^{n} \pi_i \ = \ 1 \end{gathered}$$

A good reference for portfolio optimization in discrete time is [JI87]. Portfolio optimization in continuous time is discussed in [Mer71].

## 3 Choosing a Combination of Software Validation Techniques

Software developers must choose among a variety of techniques for validating their software. We take as an example the validation of a multi-threaded software. Among the properties we might like to verify are correctness, efficiency, fairness, safety and liveness. For this example, we focus on deadlock detection, and we assume that some prior analysis has identified random testing, boundary testing, model checking, and static analysis as the set of candidate techniques. In real life, we might consider other techniques.

These techniques, differ in the following characteristics: (a) the input they need, e.g: a formal model of specifications, natural language specifications, source code, an executable, (b) the kind of resources needed, e.g. time to set up analysis, human expertise required to successfully deploy the technique, money investment to acquire the technique, equipment (hardware, software, paper, etc.), (c) the amount of each of these resources they demand, (d) degree of confidence in the results, e.g: some confidence of the absence of bugs or proof of the absence of a problem, (e) applicability to different properties.

Our goal is to allocate a limited set of resources (the budget) among the candidate techniques (various assets) in order to end up with the greatest confidence that our system is deadlock-free or free of bugs of a certain class (the return). To choose which subset of these techniques to use, and with what relative emphasis, we need to characterize them uniformly. To show how portfolio analysis might help here, we sketch the modeling of this problem qualitatively and informally. We make qualitative comparisons of the amounts and kinds of resources needed and the degree of confidence we have in the results. Table 3 summarizes this comparison.

Two of the candidate techniques – random testing and boundary testing – are forms of traditional testing. Random testing selects random test data values, whereas boundary testing selects values that are directly on, above and beneath the edges of the legal input and output values. Random testing does not require any special skills. Boundary testing requires some expertise in order to select effective boundary cases.

The confidence level in test results increases with the fraction of the system behaviors that is covered by the tests. The nondeterminism of concurrent systems leads to combinatorial growth in system behaviors and hence in the number of test cases required for a given level of coverage. The nondeterminism also complicates the design of test cases targeted to specific behaviors. In some cases, restrictions on the way the code is written may enable a specific kind of analysis; in these cases, accepting the restrictions might also pay off in reduced risk, and also a significant reduction in both the time spent on validation and the level of expertise needed to perform the validation. For example, Eraser [SBN+97] is a tool for dynamically detecting data races in lock-based multi-threaded programs. In return for using locks as the only mechanism for mutual exclusion, and using the same lock for a particular shared variable, you get data race detection for free.

Model checkers prove correctness (here, the absence of deadlocks) for finite models of a system by performing exhaustive analysis of the large but finite state space. This translates to confidence in the real system because many problems manifest themselves in small examples. Model checking requires an abstraction of the system to be specified in a special modeling language. Finding and specifying this model requires special skills, and it is often time consuming. Some model checkers do not require the use of an abstraction, e.g Verisoft [God97] but are less powerful in their verification ability (Verisoft does not handle liveness properties, and can ensure results up to a certain depth only). These model checkers still require special skills, but they eliminate the model creation task. The exhaustive analysis ensures correct results for the finite abstraction, but translating this to the real system depends heavily on the accuracy of the model or on our assessment of the probability of deeper path executions.

Static analysis techniques such as slicing and data flow analysis examine the text of a program and automatically extract information about its dynamic behavior. These techniques are well established for sequential programs and are current research for concurrent programs. For specific restricted classes of concurrent programs, static analysis can already be useful [God97]. static analysis approaches are able to evaluate all potential execution paths for certain classes of problems, it may be possible to use static analysis to demonstrate the absence of such problems [NCO98]. Setting up a static analysis currently requires a high level of specific expertise, but the reward is complete confidence in the results in case of successfully demonstrating the absence of a fault. Since the static model is overly conservative, our confidence level in case of failure to demonstrate the absence of a fault is very small. In such a case we need to further deploy some form of dynamic analysis. That is, using static analysis incurs a risk that the effort will be redundant because we discover that we need to perform dynamic analysis as well.

The available techniques can and should be combined, but some combinations make more sense than others. We concentrate here on the first-order differences. A more extensive exploration must consider technical qualities of the interaction among techniques: For example, some techniques are complementary (e.g static checking and model checking), while in other cases one technique may subsume an-

| Technique | requirements for special skills | time to set up analysis | degree of confidence in the results | restrictions on code implementation |
|---|---|---|---|---|
| Random testing | minimal | high | fairly low | none |
| boundary testing | low | high | fairly low | none |
| Model checking | very high | high | very high | none |
| static analysis | high | ? | very high | some restrictions |
| Eraser | to adhere to programming restrictions | low | high | use only locks for mutual exclusion. Same lock for a particular shared variable. |

Table 1: Comparing validation and verification techniques for deadlock detection.

other (e.g in some cases random testing may not add much value to thorough directed testing).

**A portfolio-analytic model.** The above problem of deciding how much resources to invest in each validation technique can be cast as a problem in portfolio selection, as follows. We have a budget $B$ of total resources which we want to distribute among various validation techniques. There are $n$ validation techniques at our disposal. For each technique we need to first quantify its (possibly uncertain) *benefit-to-cost ratio*, i.e., the benefit per unit resource investment. The benefit is a numerical representation of the level of confidence achieved by applying the technique, and can be modeled as a random variable with a certain distribution. The cost will reflect any special skills, or special formatting of input, or significant set-up time, or special restrictions required on the coding. This allows us to treat each technique just like a security in the portfolio problem. Now let us assume that we have some (strictly increasing and concave) utility function that maps a level of confidence to a real number. Then our problem can be stated as: find the distribution of resources among validation techniques that results in maximum expected utility. Clearly the form of the utility function is crucial to the outcome of this analysis. For instance if the utility function is linear the optimal policy is to simply invest all resources in the technique that has the highest expected benefit-to-cost ratio. For non-linear utilities the optimal policy could turn out to be one that distributes a positive amount of resources among more than one technique. The selection of the utility function and the estimation of the probabilities involved in the uncertain benefits, are difficult problems. These are usually arrived at through a series of questions posed to the people involved in the project.

## 4 Selecting a Security Portfolio

In designing a large system, various technologies are available. We illustrate how portfolio analysis might help us choose among competing technologies. Given a set of threats, the objective of the security portfolio is to select the best combination of security components within a limited budget. Since no single security component can provide protection against all possible threats, developers integrate various software designs and security technologies in an attempt to reduce the systems vulnerability to internal and external threats. Ideally, the software developer would like to evaluate candidate sets of security designs and technologies to determine whether any combination satisfies the requirement and, if so, to select the combination that provides the least risk within the specified budget. As in financial portfolios, security portfolios represent a distribution of wealth across different components that offer various degrees of effectiveness (or rates of return). Software engineers select a set of security technologies based on the threats discovered during the risk assessment. Threats represent potential attacks to the system because attackers may be able to exploit vulnerabilities. Vulnerabilities are weaknesses in the system that could allow someone to gain unauthorized access to system resources or deny authorized users access to system resources. Risk assessment identifies the system threats and vulnerabilities. Security components are chosen because they offer countermeasures against one or more threats. Software engineers consider the cost of the component, its effectiveness against specific threats, its breadth of coverage (number of threats addressed), and the value of the resource protected. Access control illustrates the dilemma for the software engineer. The most common access control to computer systems is a password. Passwords are relatively easy and inexpensive to implement, but not considered nearly as effective as Smart cards. Smart cards offer stronger protection but are more difficult to administer and cost more. As the strength of the security increases so does the cost. Suppose, for example, that a software engineer has a fixed dollar budget for system security including access control and intrusion detection. If Smart cards are chosen, there may not be enough dollars for a highly effective intrusion detection system. Alternatively, if simple password protection is chosen, then there could be sufficient dollars available to purchase an intrusion detection system and an automated back-up and recovery system. Which combination provides the best security? Portfolio analysis suggests a way for software engineers to compare alternative combinations of security solutions.

## 5 Directions for future work

While modeling the problem of allocating resources to testing and verification activities, we have to make limiting assumptions. Strictly speaking, these assumptions are not valid, but it would be interesting to know how these limiting assumptions affect the quality of decisions. The fact remains that making the connection to the well developed area of portfolio optimization allows us to bring sophisticated techniques to the area of software management. However, most of the portfolio optimization has been developed in the area of securities markets (e.g., stocks, bonds). Securities markets are generally very *liquid*, i.e., one can trade among assets freely. This is assumption is not quite valid in the context of software management decisions. For example, one cannot exchange programmers freely, i.e., a programmer with expertise in user-interfaces cannot be replaced (without any cost) by a programmer with expertise in the area of databases. This assumption clearly does not always hold, and we plan to investigate how to handle non-linear relationships between effort and benefit. Another area of interest would be gathering of metrics to quantify uncertainty. For example, how does one estimate the level of assurance for various level of investments. Finally, we want to explore the possibility of integrating portfolio optimization based decision making capabilities into software lifecycles. For explanation of topics mentioned in this section readers should refer to [Kem97, Boe81].

## References


[Boe81] B.W. Boehm. *Software Engineering Economics*. Englewood Cliffs, NJ, Prentice-Hall, 1981.

[Fis70] P. Fishburn. *Utility Theory for Decision Making*. New York, Wiley, 1970.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 174–186, Paris, France, 15–17 January 1997.

[JI87] Jr. J.E. Ingersoll. *Theory of Financial Decision Making*. Rowman and Littlefield, Totowa, NJ, 1987.

[Kem97] C.F. Kemerer. *Software Project Management: readings and cases*. Irwin, McGraw-Hill, 1997.

[Mer71] R.C. Merton. Optimum consumption and portfolio rules in a continuous time model. *J. Econ. Theory*, 3:373–413, 1971.

[NCO98] Gleb Naumovich, Lori A. Clarke, and Leon J. Osterweil. Efficient composite data flow analysis applied to concurrent programs. *ACM SIGPLAN Notices*, 33(7):51–58, July 1998.

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Operating System Review: Proceedings of the Sixteenth ACM Symposium on Operating System Principles*, volume 31, pages 27–37, St. Malo, France, October 1997. ACM Press.