# A Management Tool For Evaluation of Software Designs

Sergio Cárdenas-García and Marvin V. Zelkowitz, *Senior Member, IEEE*

*Abstract*—**The development of quality software depends upon making appropriate decisions at every stage of the life cycle. Given a design, many techniques have been developed to produce quality code from that design. However, ignored so far have been formal models to help the software manager to make appropriate implementation decisions. A model for evaluating software designs has been proposed and is based upon extending the functional model of program verification with concepts from economic decision theory. This paper briefly describes the method, and describes a prototype implementation of a tool, called *Selector*, which implements this technique.**

*Index Terms*— **Correctness, decision support systems, design evaluation, prototyping, risk analysis, software reuse.**

## I. Introduction

DEVELOPMENT of software usually consists of a set of fairly well-established processes. The "waterfall model" is typical of such tasks where an organization will develop specifications, build a design from these specifications, implement and test code based upon this design, and then maintain the resultant system during its lifetime. Much of software engineering research is concerned about improving the quality of the product at each of these stages (e.g., use of formal methods to better refine specifications; top-down design, object-oriented design, and other methods for improving the design process; various test methods for improving the verification and validation of this code, etc.).

However, few models and fewer tools have been developed to aid the software manager in the decision-making process of directing such development activities. How does one pick one design strategy over another? Which design will best meet management's objectives with respect to cost, schedule, and functionality? With increasing emphasis on improving productivity by reusing software, how to evaluate previously written components in order to determine whether they meet our current needs, whether they need to be modified, or whether they should be ignored and a new component designed?

This paper addresses this issue by describing a design-evaluation mechanism and a prototype implementation of that mechanism that can aid the software manager in making such decisions. This implementation fits into the general realm of decision-support systems as a decision aid to management in deciding an appropriate course of action. This model, derived originally from studying functional verification [7], includes an evaluation mechanism for comparing design attributes [2], with an underlying utility function model for determining the appropriateness of prototyping [3].

An automation aid can be classified as one that either lowers the expertise of personnel needed to achieve a certain level of performance, or one which raises the productivity of existing expert personnel. It is our assumption that software design is a complex process. Therefore we are assuming expert management well familiar with various design strategies, who can estimate various probabilities of certain events occurring, and who can make rational decisions based upon that behavior. Our implementation which *selects* among alternative designs (hence is called *Selector*) depends on a very knowledgeable user community which would be able to profit from such a decision-support system. We do not address in this paper mechanisms, such as expert systems, which guide less knowledgeable users with an appropriate course of action.

In the remainder of this introduction we briefly describe our implementation of *Selector* and briefly describe the information needed to invoke it. In Section II we review the underlying evaluation model of the software-development process upon which the tool is based. In Section III we give an example of its use.

### *Overview of Selector*

We are assuming that the product to be built can be described by the specification of a set of attributes like functionality, cost, schedules, and performance. We also assume that the manager has a set of potential solutions. For each solution, the manager has an ordinal ranking of how well the attribute values of the solution meet thc requircd specifications.

What decisions must the manager make? Our implementation of *Selector* will aid thc manager in the following tasks:

1) By prompting the user as to the effect each attribute has on the choice of the final product, the system will evaluate the importance of each overall solution, generate a figure of merit (called the performance level), and order the potential solutions from most favorable to least favorable.
2) Prototyping is used to provide the additional information that often is needed to make a decision. *Selector* will guide the manager in developing appropriate prototypes.

Manuscript received March 19, 1991; revised May 21, 1991. Recommended by P. A. Ng. This work was partially supported by the U.S. Air Force Office of Scientific Research through Grant 90-0031 to the University of Maryland.

S. Cárdenas-García was with the Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742. He is now with AT&T, Naperville, IL.

M. V. Zelkowitz is with the Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, MD 20742.

IEEE Log Number 9102394.

Using techniques from decision theory: (i) the risks associated with each candidate solution are evaluated; (ii) attributes which should be tested by a prototyping experiment to provide the most information are indicated; (iii) the potential payoff from using that prototype can be estimated; and (iv) the maximal amount to spend on the prototype (as opposed to making an immediate design decision with no additional information) can be computed.

3) The system can be used to allow the manager to try a series of "what if" scenarios. The manager can repeatedly enter a series of assumptions in order to determine their effect on alternative design strategies. This might provide additional data before a complex expensive implementation or prototype is undertaken.

As stated above, decision theory [4] plays a major role in the design of our evaluation policies. For example, if the problem is to travel the 240 miles from Washington, DC to New York City, possible solutions might be to drive by car, travel by train, or fly by plane. For each solution, we might have three attributes: cost of trip, time of trip, and effects of weather. A traveler could objectively rank each solution, at least with respect to cost and time, to get the relative value of each attribute (e.g., for cost we get $car < train < plane$, and for time we get $plane < train < car$).

The attribute of weather is more subjective and gets into the basis of our evaluation strategy. Assuming a trip is planned in January when icy or snowing conditions might cause dangerous driving, long delays, or canceled plane or train travel, there is no correct solution *a priori.* We need to plan a strategy which balances the various requirements of arrival at the destination at a certain time with the degree of risk the traveler wishes to undertake for each mode of travel.

Our model is based upon equilibrium probabilities. That is, the traveler is asked for a subjective determination of choosing between a guaranteed result and the probability of getting a better or worse result. In our weather example, we might ask the following question for each mode of travel:

> "For what probability $\rho$ would you be indifferent to a trip guaranteed to take 4 hours or a trip of 2 hours with probability $\rho$ and probability $1 - \rho$ of a trip of 8 hours?"

A high probability (e.g., 0.9 meaning favoring the fixed 4-hr choice) signifies risk averseness and predictability; a lower probability (e.g., 0.7 leaning more towards the second choice) signifies higher risk by choosing the 2-hr trip with a nonzero chance of a truly longer trip.

By asking such questions to an expert manager, who can answer such questions based upon either objective technical details of a specification (e.g., known performance behavior of certain specified algorithms) as well as his own management style (e.g., conservative predictable behavior versus a risky high performance style), we can build a model of the decision process.

It should be emphasized that we are not proposing a replacement for the need to make technical decisions by using our risk model. Instead, we are providing a mechanism which allows the manager to state and record reasons why certain design decisions are made, and then to clearly evaluate the effects that those decisions have on the development process.

## II. A Model for Design Evaluation

Given a specification, how does one choose an appropriate design which meets that specification? The study of formal methods and program verification only partially addresses this issue. We certainly want to produce correct programs. However, correct functionality is only one attribute our system must have. We need to schedule development to have the product built within our budget, within our available time frame, and not to use more computing resources than we wish to allocate for this task. However, how do we make such decisions?

We consider two cases for this problem. In the first, the manager knows the relevant information about trade-offs and relative importance for the various attributes of the solutions. We have developed an evaluation measure, called the *performance level,* that allows a manager to choose from among several solutions when the relative desirabilities of the attribute values are known. We call this the *certainty* case. We then extend the model to include the more realistic (and certainly harder) case where the effects of each decision are not exactly known, but we can give a probabilistic estimation for the various possibilities. We call this the *uncertainty* case. The following subsections briefly describe each model.

### A. Decisions Under Certainty

We consider correct functionality to be just one of several attributes for a solution, with multiple designs implementing the same functionality. Let's first assume that our needed functionality is specified by a function (from state to state), and also that the candidate programs are specified by functions from state to state. Let $X$ be the functionality of program $x$. Program $x$ is correct with respect to specification $B$ if and only if $X \supseteq B$ [5]. We extend this model to include other attributes as well. Since these other attributes are often concerned with nonfunctional characteristics such as resource usage, schedules, and performance, we will use the term *viable* for any solution satisfying a specification, rather than the more specific term *correctness.*

Now assume that our specifications (for both our needed software and the candidate programs) are vectors of attributes, including the functionality as one of the elements of the vectors. For example, $X$ and $Y$ are vectors of attributes that specify alternative solutions to a specification $B$. Let $S$ be a vector of objective functions, with domain being the set of specification attributes and range [0..1]. We call $S_i$ a *scaling function,* and it is the degree to which a given attribute meets its goal. We state that $X solves_S Y$ if $\forall\ i$, $S_i(X_i) \geq S_i(Y_i)$. We extend our previous definition of correctness to the following: design $x$ is viable (i.e., is correct) with respect to specification $B$ and scaling function vector $S$ if and only if $P solves_S B$. We can show that the previous definition of correctness is simply a one-dimensional example of this more general definition of viability [2].

TABLE I
ATTRIBUTES AND ORDINAL VALUES FOR FILE SYSTEMS

| Attribute | Values | | |
|---|---|---|---|
| 1) functionality | file_server(10) | | |
| 2) avg_exec_time | in_(40..80] (2) | in_(20..40] (3) | in_(10..20] (4) |
| 3) version_control | sing_version_files(1) | mult_version_files(2) | |
| 4) unit_data_access | arb_page_subrange(2) | page_(page_run) (3) | arb_file_subrange(5) |
| 5) atomic_tra_scope | sing_files_only(1) | mul_file_1_serv(2) | mul_file_mul_serv(3) |
| 6) #clients_trans | sing_client_trans(1) | mul_client_trans(2) | |
| 7) concurr_control | file(1) | page(2) | |
| 8) level_concurr | s_write_or_m_read(2) | s_write_and_m_read(3) | |
| 9) deadlock_ctrl | no_deadlock_ctrl(1) | timestamps(2) | time_limited_locks(3) |
| | deadlock_prev(5) | dlock_det_and_prev(6) | |
| 10) system_cost | in_(70..90] (2) | in_(60..70] (3) | in_(0..50] (5) |

Each attribute may not have the same importance. Assume a vector of weights $W$ called *constraints,* such that each $w_i \in [0..1]$ and $\sum w_i = 1$.

Our evaluation measure, the *performance level,* merges multiple scaled attributes and their constraints. Given specification vector $X$, scaling function $S$, and constraints $W$, the performance level is given by: $PL(X, S, W) = \sum_i (w_i \times S_i(X_i))$.

We use the performance level as our objective function: given a specification vector $B$, scaling vector $S$, constraints $W$, and potential solutions $x$ and $y$, $X$ *improves* $Y$ with respect to $\langle B, S, W\rangle$ if and only if:

1) $X solves_S B$ and $Y solves_S B$
2) $PL(X, S, W) > PL(Y, S, W)$.

We use here a very simple weighted sum to compute the performance level. Our definition of *improves* depends only upon an appropriate definition of performance level for comparing two solutions, not on the details of how the two vectors are compared. Further details of this model are given in [2].

It should be noted that the model presented in this section depends upon the solution triple $\langle B, S, W\rangle$, which is a quantitative evaluation of how well each attribute of the proposed solution meets or exceeds the minimal specification $B$. We rarely know this in practice, and this paper only assumes an ordinal ranking of the attributes—that is, one attribute value is better than another. In Section III-C we show how to evaluate $\langle B, S, W\rangle$ given only the specification and ordinal rankings for each attribute. (Table I shows an example of these ordinal rankings.)

### B. Decisions Under Uncertainty

We have so far assumed that the relative importance of each attribute is known *a priori.* However, we rarely know this with certainty. We therefore consider the following model, based upon aspects from economic decision theory [1]. The following is a brief summary of our uncertainty model, described more fully elsewhere [3].

The performance level assumes that the relative importance of each attribute is a known constant, so that the weight factors and scaling can be defined. However, this is not generally true. For example, in a program that includes a sort of a list of records, the importance of the sort algorithm itself depends upon how often it gets called and how long unsorted lists get. That is, if the list of items always remains short, then any sort algorithm will suffice, since sorting will take a negligible part of the execution overhead. In this case, any attribute value (i.e., specification) describing the sort function will have minimal effect upon the resulting program and have a very low weight. Our problem is then to modify the previous model to account for unknowns in the importance for these attribute values.

Using terminology from decision theory, the potential solutions to a specification are called *alternatives,* and the various possibilities that will determine the importance for the attribute values are *states of nature.* Each state of nature is associated with a fixed set of weights giving the relative importance of each system attribute.

We can now represent the performance level as a matrix $PL$, where $PL_{i,j}$ is the performance level for solution $i$ under state of nature $j$. As before, the performance levels give a measure of how good a system is. We can approximate this by defining the entries $PL_{i,j}$ of performance level matrix $PL$ as the payoff (e.g., monetary value) for solution $i$ under state $j$. For example, assume that we have two potential solutions $X^1$ and $X^2$, and assume we have three potential states of nature $st_1$, $st_2$, and $st_3$, which are represented as the six possible *payoffs* in the matrix:

$$PL = \begin{bmatrix} 100 & 500 & 0 \\ 300 & 200 & 200 \end{bmatrix}. \tag{1}$$

In this example, if we knew for sure that $st_2$ would be the resulting state of nature, then we would implement alternative $X^1$ (with payoff 500), and if we knew that either states $st_1$ or $st_3$ were the resultant states, then alternative $X^2$ would be most desirable. However, we may not know this beforehand.

When the probability for each state of nature can be estimated, we can use expected values to achieve an estimated performance level. Given probability distribution vector $P$, where $p_i$ is the probability that state of nature $st_i$ is true, the expected payoff for alternative $X^i$ is given by:

$$v_i = \sum_j pl_{i,j} p_j. \tag{2}$$

Use the decision rule: choose $X^i$, which maximizes $v_i$, or

$$\max_i \left( \sum_j pl_{i,j} p_j \right). \tag{3}$$

For example, if we know that the probability distribution for each state of nature in our example is $P = (0.3, 0.5, 0.2)$, we can calculate the expected payoffs as follows:

$$\begin{aligned} v_1 &= 100 \times 0.3 + 500 \times 0.5 + 0 \times 0.2 \\ &= 280 \\ v_2 &= 300 \times 0.3 + 200 \times 0.5 + 200 \times 0.2 \\ &= 230. \end{aligned}$$

We would then choose $X^1$ over $X^2$, since $280 > 230$.

*Risk Aversion:* Risk aversion plays an important role in decision making. This implies subjective behavior on the part of the software manager. We assume that the following *reasonable* behavior rule (i.e., equilibrium probability given in the introduction to this paper) is true:

- *Decomposition:* Given three payoffs $a \le b \le c$, there exists a probability $\rho$ such that the decision maker is indifferent to the choice of a guarantee of $b$, and the choice of getting $c$ with probability $\rho$ and getting $a$ with probability $1 - \rho$. We shall refer to this probability as $decomp(a, b, c)$.

For example, assume there are two techniques to solve a problem. One is fully tested, giving a guaranteed payoff of $5000, and a second new and more efficient technique promises a potentially larger payoff of $10 000 (but not completely tested), with a chance to give a payoff of only $2000. If a software manager considers using the new technique only if the chances of getting the payoff of $10 000 are larger than 80%, the probability $\rho$ is larger than 0.8. In this case the expected payoff will be $10\,000 \times 0.8 + 2000 \times 0.2 = 8400$, so the given manager is somewhat risk-averse and conservative.

Let $pl_0$ be the minimal value in our payoff $PL$, and let $pl^*$ be the maximal value. In our example, the $PL$ matrix (1), we would choose $pl^* = 500$ and $pl_0 = 0$. We decompose each $pl_{i,j}$ as $e_{i,j} = decomp(pl_0, pl_{i,j}, pl^*)$. This decomposition creates an equivalent pair of payoffs $\{pl_0, pl^*\}$, with the probability $e_{i,j}$ of getting the more desirable $pl^*$. We call the matrix formed by these elements $e_{i,j}$'s the equilibrium matrix $E$.

Any element $e_{i,j}$ will satisfy the following inequality:

$$pl_0 \times (1 - e_{i,j}) + pl^* \times e_{i,j} \ge pl_{i,j}. \tag{4}$$

The difference between the two sides of this equation reflects the manager's degree of risk-averseness. If the two sides are equal, risk analysis reduces to the expected value.

### C. Value of Prototyping

Given the various unknowns in the states of nature, the software manager may choose to get more information with a prototype so that a better final decision can be made. However, before undertaking the procedure to extract more information, one should be sure that the gain due to the information will outweigh the cost of obtaining it. Here, we try to establish an absolute boundary: what is the value of perfect information?

The best we can expect is that the results of the experiment will indicate for sure which state of nature will hold. Under this case we can choose the alternative which gives the highest performance level under the given state of nature:

$$\Phi = \sum_j p_j \times \max_i \, pl_{i,j}. \tag{5}$$

In our example, we would choose $X^1$ under $st_2$, and choose $X^2$ otherwise, resulting in performance level $\Phi$:

$$\begin{aligned} \Phi &= 0.3 \times 300 + 0.5 \times 500 + 0.2 \times 200 \\ &= 380. \end{aligned}$$

What is the value of this perfect information? Since the expected value of our performance level was computed previously as 280, the value of this information is an improvement in performance level of $380 - 280 = 100$. This is the most that we can expect our prototype to achieve and still have it cost effective.

Assume we build a prototype to test which state of nature will be true. While we would like an exact answer, since a prototype is only an approximation to the real system, the results from prototyping are probabilistic. Let $result_1, result_2, \ldots result_k$ be the possible results of the prototype. This information will be presented in a conditional probability matrix $C$, where $c_{i,j}$ represents the conditional probability of result $result_i$ given state of nature $S_j$.

Given the probabilities for each state (vector $P$) and the conditional probability matrix $C$, the marginal probability distribution (vector $Q$) for obtaining $result_i$ is given by:

$$q_i = \sum_j c_{i,j} \times p_j. \tag{6}$$

We can compute the *a posteriori* distribution matrix $P'$. $P'$ has as many rows as results from the prototype which are updated values of vector $P$. Row $i$ gives the probabilities of the states of nature, given that the result of the prototype is $result_i$:

$$p'_{i,j} = \frac{c_{i,j} \times p_j}{q_i}. \tag{7}$$

Following our example, assume that a prototype of alternative $X^2$ is planned. The planned prototype can give the following results:

$result_1$: We are satisfied with the system as presented by prototype.

$result_2$: We are not satisfied.

Assume that the conditional probabilities are estimated beforehand. For example, we estimate that if the state of nature is $st_2$, we have probabilities 0.3 and 0.7 to obtain results $result_1$ and $result_2$, respectively, from the prototype. The conditional probabilities appear in the matrix $C$ having a column for each state and a row for each result of the prototype:

$$C = \begin{bmatrix} 0.9 & 0.3 & 0.4 \\ 0.1 & 0.7 & 0.6 \end{bmatrix}.$$

From this we can calculate the probability $q_i$ for each result $i$ of the prototype, giving $Q = (0.5, 0.5)$, and the *a posteriori* distribution matrix:

$$P' = \begin{bmatrix} 0.54 & 0.30 & 0.16 \\ 0.06 & 0.70 & 0.24 \end{bmatrix}.$$

If, for example, we get $result_1$ from the prototyping study, the new expected values for alternatives $X^1$ and $X^2$ are:

$$\begin{aligned} v_1 &= 100 \times 0.54 + 500 \times 0.3 + 0 \times 0.16 \\ &= 204 \\ v_2 &= 300 \times 0.54 + 200 \times 0.3 + 200 \times 0.16 \\ &= 254. \end{aligned}$$

In this case; alternative $X^2$ should be chosen, since it gives the higher performance level.

Similarly, if we should get $result_2$ from the prototyping study, the new performance levels for alternatives $X^1$ and $X^2$ are:

$$\begin{aligned} v_1 &= 100 \times 0.06 + 500 \times 0.7 + 0 \times 0.16 \\ &= 356 \\ v_2 &= 300 \times 0.06 + 200 \times 0.7 + 200 \times 0.24 \\ &= 206. \end{aligned}$$

In this case, alternative $X^1$ is the preferred choice.

Given that our expected performance level with no information was 280 (Section II-B), we should only prototype if we gain from prototyping:

$$\begin{aligned} v_P &= 0.5 \times 356 + 0.5 \times 254 - 280 \\ &= 25. \end{aligned}$$

Since there is a positive gain $v_P = 25$, prototyping should be carried out as long as the cost to construct the prototyping study is less than this. Otherwise, an immediate decision should be made.

## III. Example of *Selector*

A prototype implementation of our evaluation strategy has been built in C and runs on SUN 3 and DEC 3100 workstations. A manager enters a table of attributes and initial constraints and then executes *Selector*. The manager is prompted for the various equilibrium probabilities, which determine the risk averseness behavior of that particular individual as well as objective characteristics of the particular solution being considered. The tool then computes the performance level for each potential solution, computes the potential gain from prototyping, and offers advice on which attribute would provide the maximum gain if it were investigated.

This example demonstrating *Selector* is based upon data presented [6]. The problem is to develop a disk file system, choosing among four potential solutions. We assume that the software manager has identified 10 attributes that are important for a solution. Table I gives the 10 attributes and their symbolic names and relative ranks—an integer between 1 and 10 in this case.

TABLE II
Specifications for Four File System Solutions

| Attribute | Basic Reqs | XDFS | CFS | FELIX | ALPINE |
|---|---|---|---|---|---|
| functionality | 10 | 10 | 10 | 10 | 10 |
| avg_exec_time | 2 | 3 | 4 | 4 | 3 |
| version_control | 1 | 2 | 1 | 2 | 1 |
| unit_data_access | 2 | 5 | 3 | 2 | 2 |
| atomic_tra_scope | 1 | 3 | 1 | 2 | 3 |
| #clients_trans | 1 | 2 | 1 | 1 | 2 |
| concurr_control | 1 | 1 | 1 | 1 | 2 |
| level_concurr | 2 | 3 | 2 | 3 | 2 |
| deadlock_ctrl | 2 | 3 | 1 | 6 | 5 |
| system_cost | 2 | 2 | 5 | 3 | 2 |

The only data the system needs initially is the relative importance of each attribute for each solution and the minimal acceptable values. For objective attributes (e.g., required performance times) quantitative values can be used; for other attributes (e.g., how easy is it to build) more subjective relative values can be used. Table II presents the data given to *Selector.* **Basic Reqs** refers to the minimal acceptable value for any solution to be viable and **XDFS, CFS, FELIX,** and **ALPINE** refer to the four proposed solutions.

### A. Eliminate Improper Solutions

*Selector* first checks that each solution is viable, and in this case discovers that solution CFS is not since it does not solve the basic requirements:[1]

```
>>Option XDFS is a viable solution.
>>Option CFS is not viable because:
  its value 'no_deadlock_ctrl' for
  attribute 'deadlock_ctrl' is inferior
  to the basic requirement
  'timestamps'.
>>Option ALPINE is a viable solution.
>>Option FELIX is a viable solution.
```

### B. Eliminate Useless Attributes and Inferior Candidates

The second stage of analysis is to determine if all of the attributes can be used to distinguish among the potential solutions. In this second step, the attribute *functionality* is eliminated from the computation of performance level, since each potential solution has the same minimal required functionality of 10 (i.e., is a functionally correct solution). Only the other nine attributes will be used to evaluate the three remaining proposed solutions in order to simplify the number of possible states of nature:

```
>>Attribute 'functionality' is useless
  because all candidates have equivalent
  values for it.
```

### C. Determine Performance Level for Each Solution

This next step is the heart of the decision process. Given the basic specifications and ordinal ranking for each solution,

[1] Output from the program will be shown with the same font style as this footnote.

TABLE III
VALUE OF SYSTEM COST ATTRIBUTE

Consider requirements B and Xmax:
B has the minimum acceptable attribute values.
Xmax has the maximum attribute values.

| | B | | Xmax | |
|---|---|---|---|---|
| ATTRIBUTE | VALUE | M-VAL | VALUE | M-VAL |
| system_cost | *in_(70..90] | * 2.00 | in_(60..70] | 3.00 |
| avg_exec_time | in_(40..80] | 2.00 | in_(10..20] | 4.00 |
| version_control | sing_version_files | 1.00 | mult_version_files | 2.00 |
| unit_data_access | page_(page_run) | 2.00 | arb_page_subrange | 5.00 |
| atomic_tra_scope | singl_files_only | 1.00 | mul_file_mul_serv | 3.00 |
| #clients_trans | sing_client | 1.00 | mul_client_trans | 2.00 |
| concurr_control | file | 1.00 | page | 2.00 |
| level_concurr | s_write_or_m_read | 2.00 | s_write_and_m_read | 3.00 |
| deadlock_ctrl | timestamps | 2.00 | dlock_det_and_prev | 6.00 |

If we modify B by replacing its value for attribute system_cost (marked with *) to have value 'in_(60..70]', what is the PERCENTAGE of improvement?:

we need to compute the scale factor and weight functions of Section II-A. The following subsection describes this process.

*Computing the Requirements:* We defined software requirements as the triple: $\langle B, S, W\rangle$. A program $X$ is considered a *viable* alternative if and only if it has the property: $\forall\, i(S_i(x_i) \geq S_i(b_i))$. However, all we have is a vector $M$ that orders the attribute values ordinally, with the property:

$$\forall\, i(M_i(X_i) \leq M_i(Y_i) \Leftrightarrow S_i(X_i) \leq S_i(Y_i)).$$

We present a method to compute $S$ and weights $W$, given $M$.

*Maximum value of an attribute:* Let $X_i^*$ be any attribute value that maximizes $M_i$.

*Value substitution of B:* Let $B^{i\leftarrow x}$ be equal to basic requirement $B$, with the $i$th attribute value $B$ substituted by $x$. Then the elements of $B^{i\leftarrow x}$ are defined as follows:

$$B_j^{i\leftarrow x} = \begin{cases} x, & \text{if } i = j \\ B_j, & \text{otherwise.} \end{cases}$$

We modify our basic requirements $B$ by replacing one of its attribute values by value $x$. The modified vector is later used to obtain the degree of desirability of that change in attribute values.

*Calculation of* $\langle S, W\rangle$: 1) Obtain $X^*$, the vector with elements $X_i^*$ for all $i$. $X^*$ is the specification of the best solution we could expect. 2) Using equilibrium probabilities to determine the improvement of using $X_i^*$ for attribute $i$ in $B$ to obtain $S_i'(x) = decomp(B, B^{i\leftarrow x}, X^*)$ for all values $x$ of attribute $i$ and for all attributes $i$. 3) Let $w_i' = S_i'(X_i^*)$ for all $i$. This gives us the importance of each maximum attribute value. 4) Calculate the vectors $W$ and $S$ as follows:

$$w_i = \frac{w_i'}{\sum w_i'} \qquad (8)$$

$$S_i(x) = \frac{S_i'(x)}{w_i'}. \qquad (9)$$

This algorithm works by deriving $S_i'$ from the equilibrium probabilities between the minimal requirement $B$ and maximum ordinal value for any attribute $X_i^*$. The weights are just the value of this maximum ordinal value, and $S$ is derived from $S'$ by appropriate scaling. It is easy to verify that $S_i(X_i^*) = 1$; $S_i(B_i) = 0$ for all $i$ and also that $\sum w_i = 1$.

Returning to our example, we apply this algorithm by asking, for each of the nine remaining attributes, what percentage improvement we would get by replacing one basic requirement with an improved value. For example, we get the following output for the *system_cost* attribute, as given in Table III.

The percentage improvement permits us to compute $decomp(B, B^{i\leftarrow x}, X^*)$. This requires that the user truly understand the effect of each design decision. Although subjective probabilities are used, it does provide a formal model of the process of making such decisions without resorting to relatively informal guesses.

After the user enters all such estimates, $\langle S, W\rangle$ is computed and the performance levels are displayed (*Scaled val.* is the computation of $S$, and *Importance* is our constraint $W$) in Table IV.

With an estimated performance level of 0.6667, the XDFS solution is the preferred choice.

### D. Prototyping Potential Solutions

If the manager is able to estimate the equilibrium probabilities, it is possible to state the trade-offs, the relative importance of the attribute values, and compute the performance levels. If there is uncertainty in some of these, then prototyping to achieve more information might be advisable. In order to simplify our analysis, we view prototypes as providing information to the manager in one of three areas:

1) The system that is prototyped provides information on how good a candidate software is from the user perspective in terms of functionality, user interface, and other user concerns. We call this a *client* prototype.
2) The software prototype provides information of how well a candidate software solution would behave in an operational environment, the performance of the interactions system/environment, the use of the environment resources, etc. We call this an *environment* prototype.
3) The prototype provides information concerning the development plan, e.g., can it be built on time, within

TABLE IV
OPTIMAL PERFORMANCE L

```
The candidates in order of decreasing PL are:

Place 1, Performance Level of 0.6667 is:
XDFS with attributes:

          NAME                    VALUE                  Rank     Scaled val.   Importance
      system_cost             in_(70..90]               2.00       0.0000        0.0901
      avg_exec_time           in_(20..40]               3.00       0.6667        0.1351
      version_control         mult_version_files        2.00       1.0000        0.0901
      unit_data_access        arb_page_subrange         5.00       1.0000        0.0901
      atomic_tra_scope        mul_file_mul_serv         3.00       1.0000        0.1351
      #clients_trans          mul_client_trans          2.00       1.0000        0.1351
      concurr_control         file                      1.00       0.0000        0.0901
      level_concurr           s_write_and_m_read        3.00       1.0000        0.0901
      deadlock_ctrl           time_limited_locks        3.00       0.2500        0.1441
      >>>>press return
Place 2, Performance level of 0.6396 is:
FELIX with attributes:
   ...
   ...  System displays other solutions and performance levels
   ...
```

budget, and with available resources like people, tools, and computers. We call this the *feasible* prototype.

Using these three categories, the following subsection describes how we identify the potential states of nature we can evaluate by prototyping a solution.

*1) Classification of Prototypes:* As stated above, we can characterize a prototype as providing information about the client's need, the execution environment, or the feasibility of the development plan. What we need to do is determine what are the possible outcomes, or states of nature, that may result if we use the prototype. We then have to evaluate how closely the state of nature we get by using the prototype reflects the actual state of nature we would get by building the final product.

A first approximation for defining the states of nature is to consider for each alternative (i.e., solution) that the world will be in only two possible states: it will be favorable or unfavorable for that alternative. Then for alternatives $X^1$ and $X^2$, we can define four states of nature: $st_1$ favorable for both $X^1$ and $X^2$; $st_2$ favorable for $X^1$, but unfavorable for $X^2$, etc. We make a more realistic approach by defining three predicates for each alternative solution.

For each alternative solution $X^i$ we define three different predicates: (i) $cli_i$. *True* if solution $X^i$ is satisfactory for client; *false* if is marginally acceptable for client; (ii) $env_i$. *True* if solution $X^i$ is satisfactory for environment; *false* if is marginally acceptable for environment; and (iii) $fea_i$. *True* if solution $X^i$ is satisfactory for feasibility; *false* if is marginally acceptable for feasibility.

A state of nature is defined as one of the $2^n$ possible combinations for the values of all the predicates. For the example in this paper, we have 3 viable solutions with 3 predicates for each resulting in $2^9 = 512$ possible states of nature. The goal is to identify the state of nature which really holds. We first try to reduce the number of states as follows:

1) Identify which predicates are relevant (e.g., if feasibility is assured and no feasibility attributes are considered, then only predicates for client and environment are considered).
2) Construct the basic set of states of nature where each state is a different combination of the values of the predicates.
3) Identify dependencies between predicates of the form: Predicate $pred_i$ ($pred$ denotes either $cli$, $env$, or $fea$) of alternative $X^i$ is *true* if predicate $pred_j$ of alternative $X^j$ is *true.* For example, we may know that if the client likes alternative $X^i$, it is sure to like alternative $X^j$.

   - The dependency $pred_i \Rightarrow pred_j$ holds iff:

   $$\forall\, k\Big(k \in set_{pred} \Rightarrow \Big(M_k(X_k^i) \le M_k\big(X_k^j\big)\Big)\Big) \qquad (10)$$

   where $set_{pred}$ is the set of attributes associated to a predicate type *pred.* If a solution has satisfactory values for some attributes, a second solution having greater or equal metric values for the same attributes is also satisfactory for these attributes.

4) Delete impossible states from the set. For each relationship found, delete the states violating this relationship.

By applying the preceding algorithm, *Selector* first asks for the potential payoffs (e.g., our elements of matrix $PL$) for 8 potential scenarios for each candidate solution. We use our previous classification of prototypes to define the scenarios. We consider that a candidate is marginal or satisfactory for each of client, environment, or feasibility. The states of nature are defined in terms of combinations of the possible scenarios for each candidate solution. (Numbers following `Payoff:` refer to user input.)

```
There are 8 scenarios for each candidate.
Enter a payoff for each combination candi-
date/scenario

If you build XDFS and this happens:
 Scenario 1:
   marginal to user
```

```
    marginal for development process
    marginal for environment
  What is the payoff for candidate XDFS/
    scenario 1?: Payoff: 200
  Scenario 2:
    marginal to user
    marginal for development process
    satisfactory for environment
  What is the payoff for candidate XDFS/
    scenario 2?: Payoff: 300

    ...
    ···  other 6 scenarios for XDFS
    ...

    ...
    ···  8 scenarios for each of other
    ···  2 solutions
```

To make an initial estimation of the probability for each state of nature, *Selector* asks for the probability that each candidate will be either satisfactory or marginal for each user, environment, and feasibility. *Selector* computes the initial probabilities for the states from the information supplied:

```
There are 3 probabilities to ask for each
candidate. If you build a candidate, what
are the chances that the system will be
satisfactory or marginal to user, for de-
velopment and to environment?

If you build XDFS, probability that it is
satisfactory to user.
  PROBABILITY: .25

If you build XDFS, probability that it is
satisfactory for development process.
  PROBABILITY: .45

If you build XDFS, probability that it is
satisfactory for environment.
  PROBABILITY: .93
    ···  similar questions for other
         2 solutions
```

Some consistency checking is provided:

```
>>>Warning: The probability that candidate
    XDFS
  is satisfactory to user (0.250)
  is smaller than the same probability
  for candidate FELIX (0.300).
  This contradicts the fact that candidate
  XDFS has attributes with better values to
  user than candidate FELIX.
  >>>>press return
```

Based upon this data, *Selector* uses equilibrium probabilities to determine the risk averseness of the user (Fig. 1). For "neutral" users, the perceived value of a solution and the expected value will be the same; however, for risk averse users, the higher curve will result (i.e., the perceived value of a solution will be less than the actual value), while for risk seeking users, the perceived value will actually be higher than the expected value.

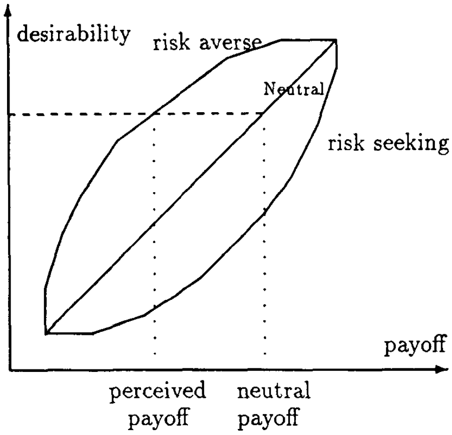


Fig. 1. Risk aversion curve and payoff.

```
Answer the following question to test if the
differences among payoffs are big enough to
produce risk effects.
What is the equilibrium probability E that
makes you indifferent to the two options:
  a) Guaranteed payoff 6.5.
  b) Probability E of getting payoff 13 and
     probability 1—E of getting payoff 0?
PROBABILITY (0.5000): .8
```

User input of 0.8 shows that there was a risk effect, so additional probabilities are needed to produce the curve represented by Fig. 1:

```
Answers to the following questions are
needed:What is the equilibrium probability
E that makes you indifferent to the two
options:
  a) Guaranteed payoff X.
  b) Probability E of getting payoff 13 and
     probability 1—E of getting payoff 0?
Enter probability E for each payoff X:
Payoff X is: 1
PROBABILITY (0.0769): .05
Payoff X is: 2
PROBABILITY (0.1538): .10
Payoff X is: 3
PROBABILITY (0.2308): .15
Payoff X is: 4

···  others
```

*Selector* will then determine: (a) which solution should be prototyped; (b) which attributes, if prototyped, are most likely to provide the most information; and (c) how much should be spent on the prototype. (In parenthesis is the expected value, meaning no additional risk. Null input from the user means that *Selector* uses this default value.)

```
CANDIDATE      THEORETICAL        PERCEIVED PAYOFF
             EXPECTED PAYOFF        (with risk)
  XDFS            3.58                 3.72
 ALPINE           8.50                 6.95
  FELIX           8.67                 6.90
Candidate ALPINE has the best perceived
payoff.
The expected payoff for this candidate is
8.50
The expected payoff with perfect informa-
tion is 9.90
The amount already invested in prototyping
is 0.00
Any future prototype should cost less than:
  9.90 - 8.50 - 0.00 = 1.40
```

In the following section we introduce a technique to determine what to prototype. This technique optimizes prototyping by suggesting a minimum sequence of prototypes (minimizing prototyping costs) to make a maximal reduction of uncertainty after each prototyping iteration.

*2) Choice of Alternatives:* The following process is used to determine for which attributes the greatest gain can be achieved by prototyping:

a) Obtain the state of nature $st_h$ that has the highest probability, that is:

$$p_h = \max\ p_i$$

b) State $st_h$ is represented by a Boolean expression of the form:

$$pred_{h,0} \wedge pred_{h,1} \wedge \cdots \wedge pred_{h,n-1}.$$

For each of the $n$ predicate values $pred_{h,k}$ (that represents a client, environment, or feasibility predicate for a candidate under state $st_h$), calculate the probability that $pred_{h,k}$ holds:

$$pvalh_i = \sum_{pred_{h,k}} p_i.$$

Probability $pvalh_i$ is the sum of the probabilities of the states consistent with predicate value $pred_{h,k}$.

c) Obtain the predicate $predicate_{pro}$ to prototype. It is the predicate that has the lowest probability for its value $pred_{h,pro}$; i.e.,

$$pvalh_{pro} = \min\ pvalh_i.$$

When there is a unique predicate having this property, that predicate is the one to be prototyped. If there are several predicates and none of them occur in a dependency ($predicate_i \Rightarrow predicate_j$), then any of them can be suggested to be prototyped. For the case of several predicates occurring inside dependencies, it is necessary to test what would happen if any of them would be prototyped. Then in a breadth first search we will choose the predicate that starts a path of prototyped predicates having minimum probability after n prototypes. For example, if our initial set of predicates having minimum probability is $\{predicate_1, predicate_2\}$ we compute for $predicate_1$:

$$pvalh_{1,j} = \min \sum_{pred_{h,1} \wedge pred_{h,j}} p_i$$

and for $predicate_2$:

$$pvalh_{2,j} = \min \sum_{pred_{h,2} \wedge pred_{h,j}} p_i.$$

If $pvalh_{1,j} < pvalh_{2.j}$, we choose $predicate_1$ as $predicate_{pro}$. We choose $predicate_2$ if $pvalh_{2,j} < pvalh_{1,j}$. If the values are equal we have to continue to consider what would happen by extending the paths another level in a similar way.

d) The kind of prototype is determined by the predicate in position *pro*. Each predicate belongs to one of the categories: *cli*, *env*, *fea*. Therefore the prototype is going to be presented to client, to environment, or will be a feasibility prototype according to the category of $predicate_{pro}$. Each predicate is associated to an alternative. The alternative associated to $predicate_{pro}$ is the one that is going to be prototyped. A creative (nonmechanical) decision remains. It consists to decide which attributes of the alternative (functionality, performance, reliability, etc.) have to be prototyped in order to decide the validity of $predicate_{pro}$.

*Selector* will now evaluate which attributes provide the most information and will suggest a potential candidate to prototype. In this example, build a prototype to see if ALPINE provides satisfactory or unsatisfactory information on environment attributes:

```
Define a PROTOTYPE to check if candidate
ALPINE is
  marginal or satisfactory for environment.
  The attributes are:
    avg_exec_time, value: in_(20..40]
    version_control,
        value: sing_version_files
    #clients_trans,
        value: mul_client_trans
    concurr_control,    value: page
    level_concurr,
        value: s_write_or_m_read
    deadlock_ctrl, value: deadlock_prev
```

At this point, the user defines his own prototype and can override the *Selector* suggestion, if desired:

```
Select a number:
  (1) XDFS.
  (2) ALPINE ← SUGGESTED.
  (3) FELIX.
selection (2): 2
The prototype is going to test if ALPINE is:
  (1) marginal or satisfactory to user.
  (2) marginal or satisfactory for devel-
      opment process.
  (3) marginal or satisfactory for environ-
      ment ← SUGGESTED.
selection (3): 3
  Maximum allowed prototyping cost:  1.40
Enter estimated COST for prototyping exper-
  iment to test if candidate ALPINE is
  satisfactory or marginal for environment:
    1.00
```

We now need to estimate how well we would trust the results of this prototype:

```
How good is the prototype?
Under the condition that the prototyping
  experiment answers 'candidate ALPINE
    satisfactory for environment', give the
    PROBABILITY that the result is correct.
    PROBABILITY (0.9000): .88

Under the condition that the prototyping
  experiment answers 'candidate ALPINE
    marginal for environment', give the
    PROBABILITY that the result is correct.
    PROBABILITY (0.9000): .88
```

The user now builds a prototype and determines how well it worked. The system can also be used to answer a series of "what if" scenarios to plan for possible contingencies.

```
Enter RESULT of the prototyping experiment
  (1) Candidate ALPINE satisfactory for
      environment.
  (2) Candidate ALPINE marginal for
      environment.
Enter result (1 or 2): 1
```

Using this information, *Selector* modifies the conditional probabilities and computes new performance levels for each candidate solution. This then becomes an iterative process with the system keeping track of how much we spent on each prototype and what our changing performance levels become. At some point, either the cost of the prototype is greater than any potential gain or the user believes he has enough information for an appropriate decision.

```
Candidate ALPINE has the best perceived
payoff.
The expected payoff for this candidate is
9.10
The expected payoff with perfect informa-
tion is 10.18
The amount already invested in prototyping
is 1.00
Any future prototype should cost less than:
  10.18 - 9.10 - 1.00 = 0.08
```

## IV. Conclusions

The ability to make appropriate design decisions depends upon explicit knowledge of the product to be developed as well as more subjective knowledge of how the product will be used. We addressed this duality by introducing a formal framework that includes risk analysis and techniques from decision theory into the software design process. The emphasis in this paper is on a quantitative measure for estimating the potential worth of prototype implementations and a prototype implementation of the mechanism. This provides the software manager with information needed to make rational decisions. Few decision support aids are currently available for this type of development environment.

We believe that a system like *Selector* can be used in two ways:

1) As a decision support system for management to be used in the process of making choices among various alternatives, and
2) As a prototyping investigative system for proposing and answering a series of "what if" scenarios. This permits various choices to be followed to their (hypothetical) conclusion before any costly implementations—either through a prototype or the full implementation—are undertaken. While the decisions generated by this tool are by no means certain, they do provide a mechanism for intelligent "guessing" the outcomes from various strategies.

Our model depends upon a risk analysis of each potential solution and aspects of decision theory to modify our evaluation strategy. The model heavily depends upon equilibrium probabilities for generating answers.

Our implementation of *Selector* hides most of the details of the model with the user simply having to estimate his own risk behavior by evaluating the equilibrium probabilities among a series of choices. While this still is far from completed, we believe we have made some first steps in bringing an algorithmic process to the top-level software decision maker. Rather than basing important decisions only on intuitive criteria, we have a formal model that can be studied and that allows for rational decision making.

Since good management consists of the ability to make choices among a set of conflicting options, there will always be a subjective component in any management model of the development process. We believe that a purely algebraic mathematical model can never accurately portray this process. Therefore the inclusion of risk analysis into the model preserves this important aspect.

## References

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

[2] S. Cárdenas-García and M. V. Zelkowitz, "Evaluation criteria for functional specifications," in *Proc. 12th IEEE/ACM Int. Conf. Software Eng.* (Nice, France), 1990, pp. 26–33.

[3] S. Cárdenas-García, "A formal framework for evaluation of multiattribute specifications," Ph.D. dissert., Dept. Computer Sci., Univ. Maryland, College Park, 1991 (Dept. Computer Sci., Univ. Maryland, College Park, Tech. Rep. 91-99, 1991).

[4] R. Charette, *Software Engineering Risk Analysis and Management.* New York: McGraw-Hill, 1989.

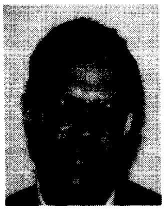
[5] H. Mills, V. Basili, J. Gannon, and R. Hamlet, *Principles of Computer Programming: A Mathematical Approach.* Newton, MA: Allyn and Bacon, 1987.

[6] L. Svobodova, "File servers for network-based distributed systems," *ACM Computing Surveys,* vol. 16, no. 4, 1984.

[7] M. V. Zelkowitz, "A functional model of program verification," *IEEE Comput.,* vol. 23, pp. 30–39, Nov. 1990.

**Sergio R. Cárdenas-García** received the B.S. degree in computer engineering in 1985 and the M.S. degree in computer science in 1986 from the Universidad Nacional de Mexico (UNAM). He received the Ph.D. degree in computer science from the University of Maryland, College Park, in 1991. His current research interests are in software evaluation, formalization of nonfunctional requirements, and software processes based on prototyping.

Dr. Cárdenas-García is a member of the Association for Computing Machinery and the IEEE Computer Society.



**Marvin V. Zelkowitz** (M'72–SM'78) obtained the B.S. degree in mathematics from Rensselaer Polytechnic Institute, and the M.S. and Ph.D. degrees in computer science from Cornell University.

He is a Professor of Computer Science at the University of Maryland, College Park, with appointments in the Department of Computer Science and the Institute for Advanced Computer Studies. He also has a faculty appointment with the Software Engineering Group of the Computer Systems Laboratory at the National Institute of Standards and Technology, Gaithersburg, MD. His research interests include software engineering, programming environments, and measurement and compiler design.

Dr. Zelkowitz is a member of ACM and is a past Chairman of ACM SIGSoft and of the Computer Society's Technical Committee on Software Engineering.