# Evaluation Criteria for Functional Specifications

Sergio Cardenas
Department of Computer Science
University of Maryland
College Park MD 20742

Marvin V. Zelkowitz
Department of Computer Science and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

**Abstract**

*Functional correctness is a technique for deriving a program and proving that this program meets its specifications. Both a program and its specifications are viewed as functions. Using techniques based upon symbolic execution and denotational semantics, a proof methodology has been developed. This current paper extends this theory of functional specifications which then permits us to model various life cycle methods in a consistent manner. Given several possible implementations for a given specification, we then develop techniques for evaluating one implementation over another.*

**Keywords**: Evaluation; Functional correctness; Program design; Specifications

## 1. Introduction

During the past fifteen years, Mills and others have been developing a theory of functional program design [Mills75, Mili86, Gannon87, Mills87]. A program is specified as a function from one domain to another and a series of steps were developed for deriving the source program from this specification. A proof methodology was developed for formally showing that a given program is correct with respect to its formally defined specifications.

The purpose of this current paper is to further extend this theory into the realm of requirements analysis. By extending the definition of program correctness, we develop a framework for discussing life cycle models and then develop an evaluation procedure to be able to compare two different program solutions for a given specification. Although it depends upon the designer subjectively assigning weights to certain attributes of the specification, it does permit an objective mechanism for evaluating competing solution strategies.

Section 2 of this paper will briefly summarize the important parts of the functional correctness methodology. Section 3 will describe our extensions to this theory and Section 4 will give examples of its applicability.

## 2. Functional Correctness

In this section we briefly summarize relevant parts of the theory of functional correctness. A *specification* $f$ is a function. A box notation $\boxed{\ }$ is used to signify the function a given string of text implements. If character string $\alpha$ represents a source program that implements exactly $f$, then $\boxed{\alpha} = f$, and we state that $\alpha$ is a *solution* to $f$.

Sequential program execution is modeled by function composition. If a sequence of statements $s = s_1; s_2; \ldots s_n$, then $\boxed{s} = \boxed{s_1} \circ \ldots \circ \boxed{s_n} = \boxed{s_n}(\ldots(\boxed{s_1}))\ldots)$. Using techniques from denotational semantics, each statement $s$ is a function from a program *state* to another *state*. Each program *state* is a function from *variables* to *values* and represents the abstract notion of data storage. Symbolic trace tables are used to derive the state functions for **if**, **while** and assignment statements. Further details are not relevant here, and the reader is directed to [Mills87] for more information on this technique.

Program design is accomplished by converting a specification function $f$, written in a LISP-like notation, into a source program $\alpha$, and then showing that $\boxed{\alpha} = f$. The specification $f$ is called the *abstract function* and the program $\alpha$ is called the *concrete design*.

A major task is the creation of data types for these functions to operate on. The concrete objects that $\alpha$ manipulates consists of the primitives of the source programming language, while the specification $f$ operates on an abstract representation of that data. Thus part of the process of showing the equivalence of $f$ and $\boxed{\alpha}$ must be to show the equivalence of the abstract and concrete representation of the data. This is done via the commuting diagram.

Let $r_d$ be a *representation function*, a function that maps all objects in a state, except $d$, into the same object, but maps concrete object $d$ into its abstract representation. If $a$ is an abstract specification for a function that operates on object $d$, and $\boxed{A}$ is its concrete realization, then we must have (i.e., Figure 1):

$$r_d \circ a \subseteq \boxed{A} \circ r_d$$

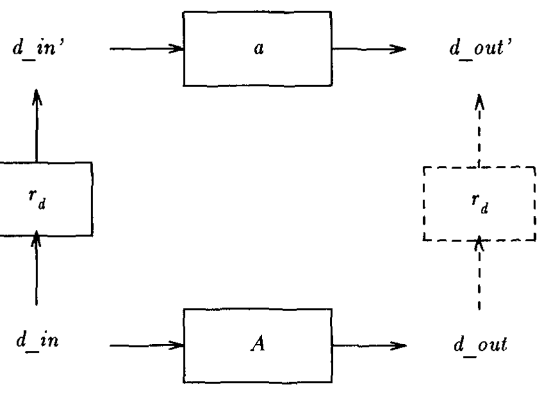

**Figure 1. Commuting Representation Diagram**

For example, the concrete realization for the abstract notion of a rational number can be a pair of integers *x_num*, and *x_denom*. The representation function $r_{rational}$ would map each variable into itself except that it would map the pair *(x_num,x_denom)* into the rational number *x_num/x_denom*. (See [Gannon87] for more details on commuting diagrams and representation functions.)

Given this functional model, we can then prove the basic theorem for functional correctness:

**Theorem:** *Functional correctness:* Program $p$ is correct with respect to specification function $f$ if and only if $f \subseteq \boxed{p}$

*Proof:* (See [Mills87, pg. 336].)

•

## 3. Specification extensions

An analysis of the commuting diagram of the previous section reveals two aspects in our current knowledge of program specifications:

(1) Specifications are in terms of functionality only. However, large systems have other considerations: cost to build, size and speed performance criteria, reliability criteria, etc. Can the model be extended to include these?

(2) With functional correctness, we might have several (actually many) feasible solutions to a given specification and any one of them might satisfy our specification constraints. Generally we usually under specify a set of requirements and any program that meets those minimal requirements is deemed acceptable.

This latter property can be made clearer via a simple example. A specification function $a$ might say to sort an array of up to 100 numbers and a concrete realization $A$ might sort up to 1,000 numbers. We would intuitively say that $A$ meets the specification $a$ (or is correct with respect to its specification) since any data in the input domain of $a$ (i.e., an array of up to 100 numbers) will be sorted even though $A$ operates on a larger domain (i.e., arrays of 101 through 1,000 numbers). The concrete realization $A$ is also specified by a more explicit specification that said to sort 1000 numbers.

Therefore, given a basic specification, there generally exists a set of specifications, each being different from the others and each satisfying the original basic specification. Each one could possibly lead to different correct implementations. We would like to be able to understand this set of specifications and to be able to compare them in order to determine which one from the set actually best meets the user's needs.

### 3.1. Functionality

Let *PU* be the ***problem universe*** of specification functions. That is, if $f$ is a specification function, $f \in PU$.

Let *SU* be the *solution universe*. *SU* represents algorithm designs. For example, if $f \in PU$ is to sort an array, then ***bubble-sort***, ***selection-sort*** and ***quick-sort*** would all be members of *SU*. The problem is to determine how well each of these actually implements a given specification. We will do this by extending the defintion of what is a correct solution. We will define a scaling function which can be used to evaluate potential implementations, show that it includes the previously given definition of correctness, and then extend it to include other attributes needed by a given program solution.

**Def:** *Scaling function:* Assume there is a function $S$ such that if $a \in PU$, then $S(a) \in [0..1]$.

Scaling function $S$ determines how good a specification we have. If $S(a)=0$, then we have a "useless" specification and if $S(a)=1$ then our specification is optimal and cannot be improved. Rarely, however, will $S$ have either extreme value. For example, in the sorting example given previously, if our "goodness" criteria is to sort as large an array as possible, if $N$ is the array we can sort by function $f_N$ and $K$ is the maximum size of memory in our machine, then $S(f_N) = N/K$ is one possible scaling function.

For a given scaling function $S$, we would like to compare two different specifications in order to determine (relative to $S$) whether one is better than the other:

**Def:** *Solves:* Let $A$ and $B$ be members of *PU*, and let $S$ be a scaling function. We say that $A$ $solves_S$ $B$ if and only if $S(A) \geq S(B)$, i.e., $A$ is an improved specification to $B$. $S$ will be omitted if it is understood which scaling function is used.

Given a program $p$, we would like to determine a specification $p'$ that exactly implements $p$.

**Def:** *Specifies:* Let $p \in PU$ and $P \in SU$ be the abstract function and concrete solution and let $r$ be the data representation function $r \ o \ p \subseteq \boxed{P} \ o \ r$. Define the exact specification $P'$ for $P$ as that function such that $r \ o \ P' = \boxed{P} \ o \ r$.

In general we cannot explicitly specify for program $P$ the exact formulation of $P'$. However, we do not need its exact definition when applying this model in many applications. It is convenient, nevertheless, to consider it when we use the model.

For example, if the representation function $r$ is one-to-one, as most applications are (e.g., each object transformed by the program represents a different abstract object defined by $r$), then we can define $P'$ as follows:

$$P' = \{(x,y) \mid \textit{for all } u \in domain(\boxed{P}),\ x=r(u),\ y=r(\boxed{P}(u))\}.$$

$P'$ is our required specification as shown by the following:

**Theorem:** Given specification $f$, program $P \in SU$ and one-to-one representation function $r$ such that $r \ o \ f \subseteq \boxed{P} \ o \ r$, then

$$r \ o \ P' = \boxed{P} \ o \ r.$$

*Proof:* First show $r \ o \ P' \subseteq \boxed{P} \ o \ r$. Let $(m,n) \in r \ o \ P'$. Then $n=r \ o \ P'(m)=P'(r(m))$. But $r(m) \in domain(P')$ means for some $q \in domain(\boxed{P})$ we have $r(m)=r(q)$ and since $r$ is one-to-one, we have $q=m$. Since $P'$ is a function, $n=r(\boxed{P}(m))$. Therefore, $(m,n) \in \boxed{P} \ o \ r$ or $(m,n) \in \boxed{P} \ o \ r$. The proof works similarly to show that $\boxed{P} \ o \ r \subseteq r \ o \ P'$.

•

With our concept of exact specifications, we can now extend the previous definition of functional correctness:

**Def:** *Correctness:* Solution $X$ is *correct* with respect to specification function $R$ and scaling function $S$ if and only if $X'$ $solves_S$ $R$.

It is important to show that we have not deviated from the previous definitions in the functional correctness methodology. Therefore, we need to show that our definition of correctness is consistent with previous formulations. We give this as the following theorem:

**Theorem:** Given specification function $R$, and solution $P$ then $R \subseteq \boxed{P}$ if and only if for an appropriate scaling function $S$, $P'$ *solves*$_S$ $R$.

*Proof:* Assume for specification function $R$ and program $P$ that we have $R \subseteq \boxed{P}$ (i.e., the original functional model definition of correctness). Then if we define scaling function $S$ to be:

$S(X)=1$ *if* $R \subseteq \boxed{X}$ *else* $0$

We then have $S(P')=1$ and $P'$ *solves*$_S$ $R$ (i.e., our new definition of correctness). Similarly, by our definition of $S$, the only $P'$ which solves $R$ is one where $R \subseteq \boxed{P}$ and hence is correct with respect to the Mills' theory.

•

Thus for simple scaling functions, we have a theory consistent with the existing model of functional correctness. However, by extending the concept of correctness, our scaling functions allow greater lattitude in determining characteristics we want in "correct" solutions.

## 3.2. Other attributes

As stated previously, functionality is insufficient as the only specification criterion. There are at least three other classes of attributes: *performance* (e.g., size, execution speed, disk usage), *reliability* (e.g., fault tolerance, accuracy, safety) and *development* (e.g., cost to build, time, personnel costs). We extend the definition of a specification function to include a vector $B$ of *basic specifications*, each $B_i$ is an *attribute* of the specification. Thus $PU$ is a set of vectors.

Similarly, we extend $S$ to be a vector of *scaling functions.* If $x \in PU$ and $y \in PU$ then $x$ *solves*$_S$ $y$ if and only if, for all $i$, $S_i(x_i) \geq S_i(y_i)$.

Given a specification vector $f$, a scaling vector $S$ and feasible programs $P$ and $Q$ such that both $r \circ f \subseteq \boxed{P} \circ r$ and $r \circ f \subseteq \boxed{Q} \circ r$, we would like to determine which solution is preferable. Obviously, if $P'$ *solves* $Q'$, or if $Q'$ *solves* $P'$, then our choice would be obvious. However, such choices rarely occur in practice. All too typically, one solution might excel on some attribute (e.g., execution speed) while the other might excel on another (e.g., low cost to build). Comparing the relative importance requires a further extension to this model.

**Def:** *Constraint set:* While there are many ways to compare two vectors of values, we will initially use a relatively simple sum of weights measure. If we have $n$ attributes, let $S$ be a vector of scaling functions. Consider a third vector of weights $W$ such that each $w_i \in [0..1]$ and $\Sigma w_i = 1$. We will call $<S,W>$ the *constraint set* for a specification.

**Def:** *Performance level:* Given a basic requirement $B$ and constraints $<S,W>$ where $S$ is a vector of scaling functions and $W$ is a vector of weights, define the *performance level PL* of $B$ relative to $<S,W>$ as $PL(B,S,W) = \Sigma\, (w_i * S_i(B_i))$.

Given a basic specification $B$ and a constraint set $<S,W>$, we can now discuss the relative merits of alternative solutions. If $x$ and $y$ are both to be feasible designs, then, at the least, they must both satisfy the specifications i.e., we must have both $x'$ *solves*$_S$ $B$ and $y'$ *solves*$_S$ $B$. In addition we would like to use the solution with the greater performance level. We call this the *improves relation.*

**Def:** *Improves:* Given a basic specification $B \in PU$, *constraint set* $<S,W>$ and designs $x$ and $y$ such that $x,y \in SU$, we state that $x'$ *improves* $y'$ with respect to $<B,S,W>$ if and only if

(1) $x'$ *solves*$_S$ $B$ and $y'$ *solves*$_S$ $B$ and

(2) $PL(x', S, W) > PL(y', S, W)$.

Note that in this paper we are using a very simple sum of weights measure in order to compute the performance level. We recognize the simplicity of this approach; however, even with such a simple model we believe that we can achieve interesting results. We plan to investigate other definitions later. For example, we can plot each of the attribute values on a circular graph and then look at the area covered by each potential solution (e.g., similar to Kiviat graphs for system performance evaluation). This and other approaches will be studied.

What is important to realize, however, is that our definition of *improves* depends only upon a definition of *performance level* to compare two solutions, not on the details of how the two vectors are compared. What follows will remain true regardless of the underlying metric used in the comparison.

## 3.3. Properties of these relations

Using the previous set of definitions, we can prove several properties of these functions. Their proofs follow easily from previous definitions.

**1.** *B solves B.*

**2.** *B improves B* is false.

**3.** If *A solves B* and *B solves C* then *A solves C.*

**4.** If *A improves B* and *B improves C* then *A improves C.*

It should be noted that if *A improves B*, it is not necessarily true that *A solves B.*

An important point to make here is that *PU*, the problem universe, is just the set of basic specifications. The constraint sets <S,W> are not part of *PU.* Given a specification in *PU*, many other specifications (which map to specific solutions in *SU*) have similar attributes, and depending upon need (as defined by the user via the constraint set) different basic specifications might be optimal.

An example using sorting explains this better. In order to sort an array, there are a sequence of potential specifications $B_1$, $B_2$, ... $B_n$, where $B_i$ specifies a sort of $i$ elements. If the requirement $B$ is to sort 100 numbers, but the user allows for solutions up to memory size (e.g., up to $k$ elements), the user can define the scaling function $S$ to be:

$S(B_i) = i/k$ for $i \in [100..k]$

$S(B_i) = 0$ for $i \in [1..99, k+1, k+2, ...]$

Given any two solutions $x$ and $y$ in *SU* that satisfies this requirement, if $x$ sorts $m$ elements and $y$ sorts $n$ elements and $m > n$ with $m, n \in [100..k]$, then $x'$ *solves B*, $y'$ *solves B*, and $x'$ *improves* $y'$. Of course, this simplistic example ignores important attributes such as execution time or memory usage which must also be considered and might affect which solution is actually more desireable, but it does show the relationship among an entire family of similar basic specifications. In Section 4, more realistic examples are given.

## 4. An Evaluation Framework

The extensions to the functional correctness theory of the previous section permits us to apply it to various specification problems. In this paper we give three different examples of its use in order to show the different classes of problems that this method can be applied to: (1) We use it to define the concept of a software prototype, and then define a life cycle model to support prototyping. We can define other life cycle processes similarly. (2) We give an example of a specification and give alternative feasible designs. We use our performance level criteria to determine the best solution from among the available ones. (3) We apply the methodology to a systems engineering problem where we determine which hardware/software solution best meets some given constraints.

### 4.1. Prototypes

A prototype is a preliminary development of a product for the purpose of evaluation. In order to be effective, it must model some aspect of the final product, but often fails to model others. For example, a software prototype will often have functionality inferior to the final product, but will be constructed at greatly reduced cost.

We can model a prototype from our basic specification methodology by considering the projection of our $k$-length scaling function to a lesser-dimensioned scaling function of fewer attributes.

**Def:** *Projections:* Let $v$ be an $n$ element binary vector of 0's (false) and 1's (true), and let $X$ be an $n$-element vector. Define a *projection* $P_v(X)$ to be those elements in $X$ corresponding to 1's in $v$. The dimensionality of the projection (and hence the number of attributes we are considering in our prototype) corresponds to the number of 1's in vector $v$.

**Def:** *Prototype:* Let $B \in PU$ be a basic specification and let $<S, W>$ be a constraint set. $X \in SU$ is a *prototype* of $B$ if and only if there is some binary vector $v$ such that $P_v(X')$ *solves*$_{P(S)}$ $P_v(B)$. We write this as $X'$ *prototypes*$_{v,S}$ $B$.

Note that the essential part of this definition states that $X'$ *solves* $B$ for some subset of attributes, so that the prototype excels at some attribute needed in the final product, but other attributes probably fail. The need to satisfy some specification attribute is an important property of prototyping, which we characterize as the *feasibility* property.

**Def:** *unfeasible:* We state that an implementation is *unfeasible* if and only if for a given specification and constraint set $<B,S,W>$, there is no $X \in SU$ such that $X'$ *prototypes* $B$. That is, if we cannot solve $B$ on a subset of its attributes, there is no way to solve it on all its attributes.

Given this definition of prototyping, we can now define a method of software development based upon prototyping. The spiral model [Boehm88] is one such example.

**Def:** *Spiral model life cycle:* The spiral model is based upon the following process algorithm:

(1) Develop an initial requirement and constraint set $<B,S,W>$.

(2) using some method, find some $B_1$ such that $B_1'$ *prototypes* $B$.

(3) Apply risk reduction in order to determine whether to continue development or terminate the project. If the result is to continue, then:

(3a) Generate a new $B_{i+1}$ such that $B_{i+1}'$ *prototypes* $B$.

(3b) Check if $P(B_{i+1}')$ *improves* $P(B_i')$.

(3c) If (3b) is true, set $i:=i+1$ else discard prototype $B_{i+1}$.

The algorithm works by generating successive prototypes, and ideally, each one is an improvement over the previous one with the process converging on a solution. If not, then step (3c) says to ignore that prototype and try an alternative strategy at that point. The crucial part of the algorithm is step (3). The system architect needs to know when to continue or terminate the process (e.g., out of funds, development time excessive, solution is unfeasible). Since the unfeasibility of any solution is generally unknown, the risk reduction heuristic is not an exact algorithm.

### Modified improves relation

It is practical to distiguish between two kinds of attributes: critical and non-critical. The non-critical attributes are those which can be absent of a solution to the problem. That is, if position $i$ of basic requirement $B$ corresponds to a non-critical attribute, then the vector $B$ (basic requirement) is defined such that $S_i(B_i) = 0$. It should be pointed out that non-critical requirements can still make a solution more desirable. Consider solutions $x$ and $y$ and assume that both are identical except that $x$ provides non-critical attribute $i$ in some degree and $y$ does not. That is:

$$S_i(x_i) > 0$$
$$S_i(y_i) = 0$$

Clearly, since both are solutions, we have that $x$ *improves* $y$.

A critical attribute has an associated critical value. Any proposed solution should have at least this critical value for the attribute. Again, the critical values are stored in the basic requirement, and then, if position $j$ of basic requirement $B$ corresponds to a critical attribute, $0 < S_j(B_j) \leq 1$. Any proposed design is considered as a solution if it provides this critical value for the attribute. (This is guaranteed from the definition of relation *solves*.) In general, apart from this distinction, critical attributes will have a larger weighting factor in its *performance level* than non-critical attributes.

A modification to the relation *improves* allows a greater distinction between critical and non-critical attributes. Let $v$ be a projection vector with 1's corresponding to the critical attributes.

**Def:** *Modified improves relation.* Given a basic specification $B \in PU$ constraint set $<S, W>$; projection vector $v$ of critical attributes; and designs $x$ and $y$ such that $x,y \in SU$ we state that $x'$ *improves*$_v$ $y'$ with respect to $<B,S,W>$ if and only if

(1) $x'$ *solves*$_S$ $B$ and $y'$ *solves*$_S$ $B$

(2) $PL(x',S,W) > PL(y',S, W)$

(3) $x'$ *prototypes*$_{v,S}$ $y'$.

### 4.2. Software design example

As shown previously, given a basic specification, there are usually multiple programs that can implement that specification, but from the user's perspective, some solutions may be more desireable than others. In the example that follows, a basic sort function is specified. Five alternative application areas are given (labeled $P_1$, ... $P_5$), each one specifies a sort program under slightly different constraints. Simply by changing the constraint set, we can show that different solution algorithms are more optimal for each such problem specification.

## Basic Specification

The following vector ($B_1$, ...,$B_6$) defines the attributes we want all 5 solutions to possess. These can be considered to be the minimal set of attributes we need. The constraint set will define an evaluation procedure to show how each feasible algorithm meets or exceeds this specification:

*attribute*$_1$ — *Functionality*— What the program is to compute. $B_1(x)=y$. Ordered$(y)$ and $y$ = Permutation($x$).

*attribute*$_2$ — *Average execution time*. $B_2$=50 N logN for sort input of size *N*.

*attribute*$_3$ — *Worst case execution time*. $B_3=20\ N^2$.

*attribute*$_4$ — *Space used for data*. $B_4=2N$.

*attribute*$_5$ — *Stability*— Is the data reordered for equal keys?. $B_5$= *false*. (Stability is not required.)

*attribute*$_6$ — *Source program size*— Basic algorithm size. $B_6$=50.

## Problem specifications

Given the above basic specification, we now give 5 similar applications of sorting. The 5 application areas are:

**P$_1$**. Execution times are most critical and space is somewhat important. Stability and source code size do not matter.

**P$_2$**. This is similar to $P_1$ except stability of the algorithm is somewhat important.

**P$_3$**. Average execution time is quite important and all other attributes have equal weight.

**P$_4$**. This might represent a solution for a small machine. Space and source program size are critically important and average execution time is ignored.

**P$_5$**. All attributes have equal weights.

We will show that by using different constraint sets, we can obtain a different optimal solution for each of the above 5 problems.

## Constraint Sets

In considering the constraint set $<S, W>$, to simplify the analysis we will use the same scaling function $S$ for all 5 problems. Changing $W$ will enable us to choose among feasible designs.

## Scaling Functions

The choice of scaling function is a subjective choice of the system architect. The following scaling functions have been chosen to give higher weights to more optimal values for each attribute. The following six scaling functions will be used:

$S_1$ — *Functionality*. Function either works or doesn't work.

$$S_1(x)= \begin{array}{ll} 1.00 & x=B_1 \\ 0.00 & \text{otherwise} \end{array}$$

$S_2$ — *Minimize average execution time.*

$$S_2(x)= \begin{array}{ll} 0.00 & x \in [40N^2...) \\ .25 & x \in [40N\log_2N..40N^2) \\ .50 & x \in [35N\log_2N..40N\log_2N) \\ .75 & x \in [20N\log_2N..35N\log_2N) \\ 1.00 & x \in [0..20N\log_2N) \end{array}$$

$S_3$ — *Minimize worst case execution time.*

$$S_3(x)= \begin{array}{ll} 0.00 & x \in [100N^2...) \\ .25 & x \in [10N^2..100N^2) \\ .50 & x \in [2N^2..10N^2) \\ .75 & x \in [40N\log_2N..2N^2) \\ 1.00 & x \in [0..40N\log_2N) \end{array}$$

$S_4$ — *Minimize space used.*

$$S_4(x)= \begin{array}{ll} 0.00 & x \in [3N...) \\ .25 & x \in [2N..3N) \\ .50 & x \in [1.5N..3N) \\ .75 & x \in [1.1N..1.5N) \\ 1.00 & x \in [0..1.1N) \end{array}$$

$S_5$ — *Stability.*

$$S_5(x)= \begin{array}{ll} 1.00 & \text{x is stable.} \\ 0.00 & \text{x is not stable.} \end{array}$$

$S_6$ — *Minimize source program size.*

$$S_6(x)= \begin{array}{ll} 0.00 & x \in [200..) \\ .25 & x \in [70..199] \\ .50 & x \in [40..69] \\ .75 & x \in [30..39] \\ 1.00 & x \in [0..29] \end{array}$$

## Weights

For each of the five problems previously specified (e.g., $P_1$, ..., $P_5$), weights for each scaling function must be specified. These are sumarized by Table 1 and reflect the relative importance for each attribute given in the statement of the specification. In all cases, correct functionality (*attribute*$_1$) will have a comparable weight of .50. These different weights will result in different algorithms as being most appropriate.

## Feasible Designs

Five feasible sorts are given as potential solutions to the basis specifications (from [Knuth73, pg. 381]). Their characteristics are given by Table 2. The five sorts are: Straight insertion, Quicksort, Heapsort, List/Merge sort and Distribution Counting sort.

| | **P$_1$** | **P$_2$** | **P$_3$** | **P$_4$** | **P$_5$** |
|---|---|---|---|---|---|
| attribute$_1$–Function | .50 | .50 | .50 | .50 | .50 |
| attribute$_2$–Avg. Exec.time | .25 | .25 | .30 | .00 | .10 |
| attribute$_3$–Worst exec.time | .10 | .10 | .05 | .10 | .10 |
| attribute$_4$–Space | .15 | .10 | .05 | .20 | .10 |
| attribute$_5$–Stability | .00 | .05 | .05 | .00 | .10 |
| attribute$_6$–Source code | .00 | .00 | .05 | .20 | .10 |

**Table 1. Weights for 5 Problem Specifications**

| | Str. Insrt. | Quicksort | Heapsort | List/Merge | Dist. Count |
|---|---|---|---|---|---|
| $attribute_1$ | sort | sort | sort | sort | sort |
| $attribute_2$ | $2N^2+9N$ | 16.8NlogN-1.7N | 33.3NlogN+.2N | 20.8NlogN+4.9N | 22N+10010 |
| $attribute_3$ | $4N^2$ | $\geq N^2$ | <37.5NLogN | 20.8NlogN | 22N+10010 |
| $attribute_4$ | N | N+2$\epsilon$ logN | N | N(1+$\epsilon$) | 2N+1000$\epsilon$ |
| $attribute_5$ | 12 | 63 | 30 | 44 | 26 |
| $attribute_6$ | True | False | False | True | True |

**Table 2. Characteristics for sort algorithms (from [Knuth73, pg. 381])**

**Analysis**

By applying the scaling functions $S_i$ to each of the 5 feasible algorithms of Table 2, we get the matrix of Table 3.

| Attr | Str. | Quick | Heap | Lst/Mer | Dist. | $S_i(B_1)$ |
|---|---|---|---|---|---|---|
| 1 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| 2 | .25 | 1.00 | .75 | .75 | 1.00 | .25 |
| 3 | .50 | .50 | 1.00 | 1.00 | 1.00 | .25 |
| 4 | 1.00 | .75 | 1.00 | .75 | .25** | .50 |
| 5 | 1.00 | 0.00 | 0.00 | 1.00 | 1.00 | 0.00 |
| 6 | 1.00 | .50 | .75 | .50 | 1.00 | .50 |

** — Fails *solves* relation

**Table 3. Scaled values for each specification**

We would like to choose that algorithm that solves the basic specification and has the highest performance level for each of our 5 problem specifications. From Table 3, we note that all algorithms, except Distribution Counting, $solves_S$ the basic specification $B$ (i.e., in Distribution Counting $S_4$=.25 which is less than the minimal .50 requirement in $attribute_4$ of $B$). Therefore we can compute the performance level for each of the remaining 4 algorithms for each of the 5 problem areas; as given in Table 4.

| | $P_1$ | $P_2$ | $P_3$ | $P_4$ | $P_5$ |
|---|---|---|---|---|---|
| Str. Insrt. | .76 | .76 | .75 | **.95** | .88 |
| Quicksort | .91 | .88 | **.89** | .80 | .78 |
| Heapsort | **.93** | .89 | .86 | **.95** | .85 |
| List/Merge | .90 | **.91** | **.89** | .85 | **.90** |

**Table 4. Performance Level for each solution.**

For each column of Table 4 (e.g., the problem specification), the maximum value corresponds to that algorithm that best meets the specification according to the given constraint set. We can therefore read off the appropriate solutions directly from Table 4:

**$P_1$**. Heapsort is the appropriate solution.
**$P_2$**. List/Merge is correct.
**$P_3$**. Either Quicksort or List/Merge should be used.
**$P_4$**. Either Straight insertion or Heapsort is correct.
**$P_5$**. List/Merge is appropriate.

Thus, given the same basic specification, we get 5 similar problem statements that lead to quite different solution strategies.

## 4.3. System design example

As a second more complex design example, we use this evaluation methodology for the determination of an appropriate file system for a computer system. The goal is to specify a system as a series of attributes (10 in this case), rate the importance of each (the weights) and then give four proposed solutions. Our evaluation methodology will be used to choose among the four feasible designs.

The 10 attributes are as follows. The basic requirements and weights for each attributes are given in Table 5:

*Attribute 1 (Functionality).* Data sharing among multiple machines. The system provides a full-scale file system. It controls concurrency accesses to files, determines access rights, enforces access restrictions and provides directory structures that recognize textual names and support grouping of files. It provides recoverable files. It allows the use of diskless workstations.

*Attribute 2 (Average execution time of elementary operations).* The elementary operations considered are: create-file, read-data, write-data, delete-file.

*Attribute 3 (Version control).* Possibility to handle committed versions of a file (e.g, for auditing purposes).

*Attribute 4 (Size of unit of data access).* Fraction of a file that can be transferred to and from clients as a result of a single read-data or wite-data operation.

*Attribute 5 (Atomic transaction scope).* Number of files that an atomic transaction can access in one or multiple servers.

*Attribute 6 (Number of clients per transaction).* Degree of distributivity of a transaction in the system.

*Attribute 7 (Concurrency control).* Granularity of concurrency control.

*Attribute 8 (Level of concurrency).* Number of readers and writers allowed for individually controlled items.

*Attribute 9 (Deadlock control).* Degree of control of the deadlock problem.

*Attribute 10 (Relative cost).* Estimated cost to build the system.

| Attr. | Description | Wt. | Basic Req. |
|---|---|---|---|
| 1 | Functionality | .500 | full functionality |
| 2 | Avg. exec. time of el. operation | .200 | 60 |
| 3 | Version control | .050 | single–version |
| 4 | Size of unit of data access | .070 | page |
| 5 | Atomic trans. scope | .035 | single files |
| 6 | Num. of clients per trans. | .025 | single–client |
| 7 | Concurrency control | .025 | file |
| 8 | Level of concurrency | .025 | single writer, multiple readers |
| 9 | Deadlock control | .030 | timestamps |
| 10 | Relative cost | .040 | 80 |

**Table 5. Attributes, weights and basic requirements**

Functionality was considered the most important attribute (weight of 0.5). A performance index (execution time of elementary operations) was considered the next most important. Other attributes were considered less important and received lower weights.

The scaling functions determine how each proposed solution meets each attribute's goal. The scaling functions we use for this example are:

$S_1$ *(Functionality).* Function either works or doesn't work.

| | | |
|---|---|---|
| $S_1(x)=$ | 1.00, | $x=B_1$ |
| | 0.00, | otherwise |

$S_2$ *(Minimize average execution time).*

| | | |
|---|---|---|
| $S_2(x)=$ | 0.00, | x∈[80..) ms |
| | .25, | x∈[40..80) |
| | .50, | x∈[20..40) |
| | .75, | x∈[10..20) |
| | 1.00, | x∈[0..10) |

$S_3$ *(Version control).*

| | | |
|---|---|---|
| $S_3(x)=$ | 0.00, | x=single–version files |
| | 1.00, | x=multiple–version files |

$S_4$ *(Unit of data access).*

| | | |
|---|---|---|
| $S_4(x)=$ | 0.00, | x=file |
| | .25, | x=page(page run) |
| | .50, | x=arbitrary file subrange |
| | .75, | x=arbitrary page subrange |
| | 1.00, | x=record |

$S_5$ *(Atomic transaction scope).*

| | | |
|---|---|---|
| $S_5(x)=$ | 0.00, | x=include single files only |
| | .50, | x=multiple files and one server |
| | 1.00, | x=multiple files in multiple servers |

$S_6$ *(Clients in a transaction).*

| | | |
|---|---|---|
| $S_6(x)=$ | 0.00, | x=single–client transactions |
| | 1.00, | x=multiple–client transactions |

$S_7$ *(Concurrency control).*

| | | |
|---|---|---|
| $S_7(x)=$ | 0.00, | x=file |
| | .50, | x=page |
| | 1.00, | x=dynamically variable |

$S_8$ *(Level of concurrency).*

| | | |
|---|---|---|
| $S_8(x)=$ | 0.00, | x=single writer or single reader |
| | .50, | x=single writer or multiple readers |
| | 1.00, | x=single writer and multiple readers |

$S_9$ *(level of deadlock control).*

| | | |
|---|---|---|
| $S_9(x)=$ | 0.00, | x=no deadlock control |
| | .20, | x=ordering by timestamps |
| | .40, | x=time–limited locks |
| | .60, | x=deadlock detection |
| | .80, | x=deadlock prevention |
| | 1.00, | x=deadlock detection and prevention |

$S_{10}$ *(Relative system cost).*

| | | |
|---|---|---|
| $S_{10}(x)=$ | 0.00, | x∈[90..) |
| | .25, | x∈[70..90) |
| | .50, | x∈[60..70) |
| | .75, | x∈[50..60) |
| | 1.00, | x∈[0..50) |

The next step is to consider alternative solutions. In this example, four solutions are proposed and are nominally based upon four published algorithms [Svobodova84]. Solution 1 corresponds to XDFS (Xerox distributed File System), Solution 2 corresponds to CFS (Cambridge File System), Solution 3 to FELIX (file server developed at Bell–Northern Research) and Solution 4 to ALPINE (file system developed at Xerox Palo Alto Research Center). Some of the attributes were fixed arbitrarily only for illustration of the evaluation technique. The evaluation of the scale functions for each one of the solutions appears in Table 6.

| | Solution 1 | Solution 2 | Solution 3 | Solution 4 |
|---|---|---|---|---|
| $S_1$ | 1 | 1 | 1 | 1 |
| $S_2$ | 0.50 | 0.75 | 0.75 | 0.50 |
| $S_3$ | 1 | 0 | 1 | 0 |
| $S_4$ | 0.75 | 0.50 | 0.25 | 0.25 |
| $S_5$ | 1 | 0 | 0.50 | 1 |
| $S_6$ | 1 | 0 | 0 | 1 |
| $S_7$ | 0 | 0 | 0 | 0.50 |
| $S_8$ | 1 | 0.50 | 1 | 0.50 |
| $S_9$ | 0.40 | 0 | 1 | 0.80 |
| $S_{10}$ | 0.25 | 1 | 0.50 | 0.25 |

**Table 6. Scale function values for file systems**

In order to determine an appropriate solution, use the *performance level* computation in the original *improves* relation to determine that Solution 1 is most appropriate for this application, with Solution 3 a close second (See Table 7).

|  |  |
|---|---|
| Solution 1 | **0.805** |
| Solution 2 | 0.732 |
| Solution 3 | 0.802 |
| Solution 4 | 0.724 |

**Table 7. Weighted sums for each solution** (*performance level*)

## 5. Conclusions

In this paper we have looked at an aspect of functional specifications and developed an evaluation criteria for comparing solutions to a given set of requirements. Using this model we can also develop a framework for classifying and describing various life cycle models.

The ability to formalize these concepts is crucial for analyzing software requirements quantitatively. We must be able to classify accurately various strategies before the science of requirements analysis progresses as far as other program development concepts – such as compiler design and programming methodology – have progressed.

While the work is still preliminary, we believe that we have an interesting model that can easily be expanded upon. Various definitions of *performance level* need to be studied in order to best approximate true system design. In addition, as shown by the various examples, the model is applicable in various application domains. Other such examples need to be developed.

## 6. Acknowledgements

This research was supported in part by grant AFOSR-87-0130 from the Air Force Office of Scientific Research to the University of Maryland.

## 7. References

[Boehm88] Boehm B., A spiral model of software development and enhancement, *IEEE Software* 5, 3 (1988) 61–72.

[Gannon87] Gannon J.D., R. G. Hamlet, H. D. Mills, Theory of Modules, *IEEE Trans. on Soft. Eng.* 13,7 (1987) 820–829.

[Knuth73] *Knuth, D., The Art of Computer Programming, Vol. III,* Addison-Wesley, (1973).

[Mili86] Mili A., J. Desharnais, J. R. Gagne, Formal models of stepwise refinement of programs, *ACM Computing Surveys* 18, 3 (1986) 231-276.

[Mills75] Mills, H.D., The new math of computer programming, *Comm. of the ACM* 18,1 (1975) 43–48.

[Mills87] Mills H.D., V. R. Basili, J.D. Gannon and R. G. Hamlet, *Principles of Computer Programming: A mathematical approach*, Wm. C. Brown (1987: Dubuque, IA).

[Svobodova84] Svobodova, L. File Servers for Network-Based Distributed Systems, *ACM Computing Surveys* 16, 4, (December 1984).