56-61

136136

P-9

N93-17167

14

An Improved Classification Tree Analysis of High Cost Modules Based Upon an Axiomatic Definition of Complexity

Jianhui Tian
Soft. Eng. Process Group
IBM Canada Laboratory
North York Ontario, Canada

Adam Porter
Computer Science Dept.
University of Maryland
College Park, Maryland

Marvin V. Zelkowitz
Inst. for Advanced Computer Studies
and Computer Science Dept.
University of Maryland
College Park, Maryland

Abstract

Identification of high cost modules has been viewed as one mechanism to improve overall system reliability, since such modules tend to produce more than their share of problems. A decision tree model has been used to identify such modules. In this current paper, a previously developed axiomatic model of program complexity is merged with the previously developed decision tree process for an improvement in the ability to identify such modules. This improvement has been tested using data from the NASA Software Engineering Laboratory.

1 Introduction

Identification of high cost modules has been viewed as one mechanism to improve overall system reliability, since such modules tend to produce more than their share of problems. In order to identify such modules, Selby and Porter [2, 3] developed a decision procedure based upon decision trees. With their technique, which we call Classification Tree Analysis (CTA), they showed on a set of 16 large-scale programs containing over 4700 modules obtained from the NASA Software Engineering Laboratory, that they could identify which subset of the 74 measures obtained from each module would produce good estimators of high-cost modules.

Recently Tian and Zelkowitz [4] developed an axiomatic model of program complexity. Based upon this model, the 74 measures kept on each of the 4700 modules could be reduced to only 18 measures that represented valid complexity measures. Using these 18 measures, the decision tree process results in an improvement over the original Selby-Porter method.

In this paper we will first describe the original decision tree process, we then summarize the axiomatic complexity model, and then demonstrate that we can improve on the previous model in identifying high-cost modules.

2 Classification Tree Analysis

In a series of earlier studies by Selby and Porter, a technique called classification tree analysis (CTA) was used to identify high cost components. Of critical importance to CTA is the selection of measures (or attributes) to construct the classification tree.

We define a high cost component as one in the uppermost quartile (i.e., 25 percent) relative to past data. The rationale for this definition is the so called "80:20 rule", which states that about 80 percent of a software system's cost is associated with roughly 20 percent of the system.

A classification tree is essentially a decision tree that branches on the range of values according to a measure at an internal node repeatedly until a component can be identified as high or low cost, or until all measures are exhausted.

The classification tree method that was used, called the classification paradigm, consists of the following three integral parts:

- **Classification tree generation** is the central activity of constructing classification trees and preparing them for analysis and feedback;

- **Data management and calibration** are the activities that retain and manipulate historical data and tailor classification tree parameters to the development environment; and
- **Analysis and feedback** is the part that leverages the information resulting from the tree generation by applying it in the development process. The central piece of the application of classification tree is to develop remedial plans and take corrective actions.

2.1 CTA Method

The goal is to predict high cost modules in the current project with *high* cost being interpreted as the highest quartile. The historical data (or training set), consisting of one project immediately preceding the current one, are grouped into quartiles according to a measure's value, with all measures being considered.

Starting from the root, a measure is selected to separate modules into four subsets associated with each arc. The number to the left of an arc is the lower (inclusive) bound and the number to the right is the upper (non-inclusive) bound for the subset according to the measured value. So we have four subsets (quartiles).

A set of modules associated with an arc is positively identified if more than a threshold (termination criterion) of modules are in the highest quartile of cost, and it is represented in the tree as a terminal node marked with a "+" sign. A set can be negatively identified similarly, and represented correspondingly by a "−" sign. If a set cannot be either positively or negatively identified, another measure is selected to further classify these modules into finer subsets. This process continues until either all modules are identified or all measures are exhausted without being able to make such a determination. In the latter case, the terminal node is marked with a "?" sign, representing that CTA can not make a prediction for modules in this set.

Notice that the generation of the classification tree depends solely on the training set and various parameters selected for the technique. The current project will only use the tree but not affect the structure of the tree.

	Modules				
	m_1	m_2	m_3	m_4	m_5
cyclomatic complexity	3	8	13	30	45
module+function call	8	40	7	3	12
operators	30	18	10	33	58
module calls	3	4	3	0	5
prediction	−	?	−	−	+
actual	−	−	+	−	+

Table 1: Predicting High Cost Modules

As an example, consider the sample (fictitious) test data of Table 1, and the classification tree in Figure 1. This test set includes 5 modules and 4 measures. In this case, the CTA method predicts 3 out of 4 modules correctly (it misses module m_3) and is unable to classify module m_2 through the classification tree. For example, module m_5 follows the right most branch from the root (cyclomatic complexity of m_5 is greater than 26) and again follows the right most branch from there (operator counts of m_5 is greater than 34). We can finally predict it to be of high cost because its module call counts falls between 4 and 10.

2.2 CTA Cost

There are two types of cost associated with the CTA technique: the cost of building classification trees and the cost of using them. The former is determined by the factors: 1) the CTA parameters, 2) the size of the available measure pool where measures are to be selected, and 3) the implementation efficiency of the CTA supporting tools. For the latter cost factor, the tree size is a good measure. Because the classification trees we are studying have fixed structure (there are 4 branches from every internal node), we can effectively capture the cost of using classification trees by counting the number of internal nodes for them.

2.3 CTA Performance

According to the match between CTA predictions and actual cost data for the modules in a test set, various performance measures can be defined:

Coverage: The percentage of modules (either positively or negatively) identified;

Accuracy: The percentage of correct matches between predictions and actual data;

Comsistency: The percentage of predicted high cost modules who are actually high cost. High consistency

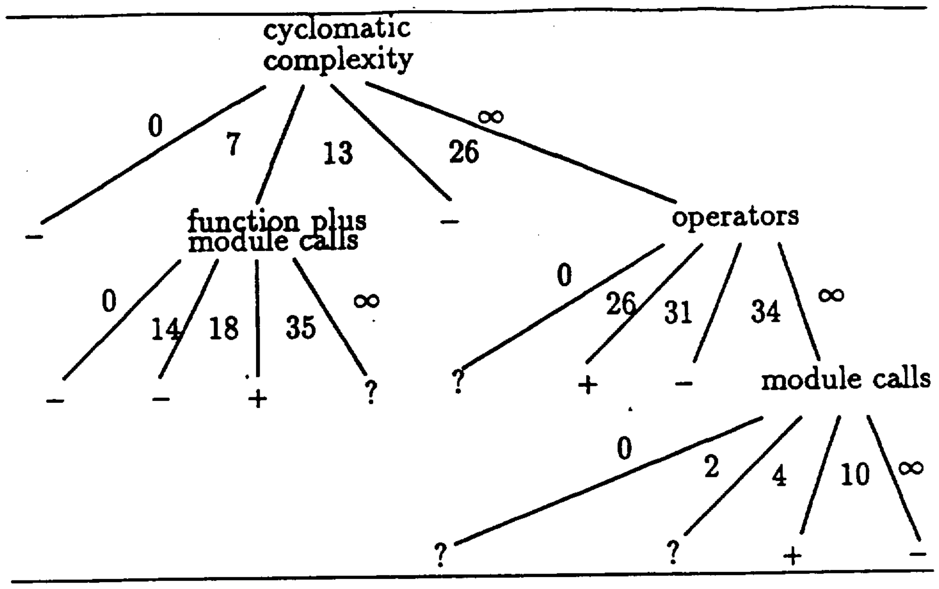


Figure 1: Component Classification Tree

indicates less "false alarms;" and

Completeness: The percentage of actual high cost modules predicted correctly by CTA. It reveals the power of CTA to uncover high cost modules.

3 Axiomatic Program Complexity

Most program complexity studies define complexity as a numeric comparison between any two programs. However, we have come to realize that some programs are inherently incomparable. For example, it makes litttle sense to compare the complexity between a payroll system and a real-time emission control system in a car. They each come from radically different application domains.

Instead we view *complexity* as a partial ranking among the set of programs and a *complexity measure* as a function applied to specific programs as an approximation of the attribute we are trying to determine. The following summarizes this model [4].

3.1 Axiomatic model

Consider a program as a hierarchy of modules consisting of instructions, data, and the underlying execution control mechanism. We initially limit ourselves to a Pascal-like nested scope sequential control language. Programs are represented by their abstract syntax trees:

- **U** represents the set of all programs.
- $AST(P)$ represents a binary abstract tree representation for program P. The root node of program P is given by $root(P)$, the left subtree of P is $left(P)$ and the right subtree of P is given by $right(P)$.
- For programs P and Q, $IN(P,Q)$ is true if P is a subprogram of Q (i.e., $AST(P)$ is a subtree of $AST(Q)$).
- If $IN(P,Q)$ is true, then $dist(P,Q)$ represents the path length in order to go from $root(P)$ to $root(Q)$.
- P with all free occurrence of x replaced by y not in P is denoted as P^x_y. We use P^α_β to mean the renaming is carried out for all corresponding one-to-one pairs in lists α and β, where

$$(var(P) - \alpha) \cap \beta = \emptyset$$

($var(P)$ is the variable list of program P).

A *complexity ranking* $\mathcal{R}$ is a binary relation on the set of programs. The complexity ranking between programs P and Q is $\mathcal{R}(P,Q)$. We interpret $\mathcal{R}(P,Q)$ as P being no more complex than Q. P and Q are *comparable*, denoted $\mathcal{C}(P,Q)$, if either $\mathcal{R}(P,Q)$ or $\mathcal{R}(Q,P)$ holds, i.e., $\mathcal{C}(P,Q)$ iff $\mathcal{R}(P,Q) \vee \mathcal{R}(Q,P)$.

A *complexity measure* $\mathcal{V}$ is a function that maps every program into a vector of real numbers: $\mathcal{V} : \mathbf{U} \rightarrow \mathbb{R}^n$.

Although simple definitions, we are immediately confronted by a difficult problem:

Theorem T1: There exist complexity rankings that are undecidable.[1]

Although the general problem of complexity ranking is undecidable, many practical rankings are not. In what follows we restrict ourselves to these more practical rankings.

Axiom A1: $(\forall P, Q)\ (\boxed{P} = \boxed{Q} \Rightarrow \mathcal{C}(P,Q))$ where $\boxed{X}$ is the function of program X.

Given programs P and Q, the problem of $\boxed{P} = \boxed{Q}$ is unfortunately also undecidable. This axiom, then, is at the center of the problem of developing effective complexity measures on real programs. We certainly want to be able to compare equivalent programs in order to determine which is best; however, undecidability says that we cannot always do this. It is for this reason that most complexity measures have not achieved significant breakthroughs since the underlying models are rarely comparable. However, in many practical applications, such as described above, we know or can assume that two given programs have the same or similar functionality.

Because of this, in practice we often use a weaker form of this axiom that only addresses the similarity of two programs:

Axiom A1': $(\forall P, Q)\ (\boxed{P} \approx \boxed{Q} \Rightarrow \mathcal{C}(P,Q))$.

A program in general consists of many hierarchically related components. As a result, we require that a program must be comparable with a subpart of itself.

Axiom A2: $(\forall P, Q)\ (IN(P,Q) \Rightarrow \mathcal{C}(P,Q))$

[1]Axiom and theorem references are keyed to [4], which also contains the proofs of the theorems. Some of the theorems given in that paper are not relevant to this present discussion and hence are not listed here.

Axiom **A2** brings up the intuitive notion that we would like complexity to increase as programs become larger, i.e., if P is a component in Q $(IN(P,Q))$, then P is no more complex than Q. We left this out because there are cases where the opposite is true. Consider Q formed from P by addition of easily recognizable keywords or tags; Q might be more readable, thus easier to maintain as a result. Another case is that loops are often more easily understood if they include their initialization code than if presented without it.

Contextual information might help to reduce the complexity of composite programs. But the degree of the reduction must be limited, otherwise infinitely large programs paradoxically might be the simplest. On the other hand, a periodic function such as *cosine(x)* as the complexity of a program, where x is some size measure of a program P, is clearly not acceptable. As a general trend, then, adding components must result in a more complex program:

Axiom A3:
$(\exists K \in \mathbb{N})(\forall P, Q)((IN(P,Q) \wedge (dist(P,Q) > K)) \Rightarrow \mathcal{R}(P,Q))$

Since our goal is to compare the complexity of two different programs, define a predicate $\mathcal{T}$ such that $\mathcal{T}(\mathcal{V}(P), \mathcal{V}(Q))$ is true if program P is no more complex than program Q. For $\mathcal{V}$ into $\mathbb{R}$, we have the obvious definition that $\mathcal{T}(\mathcal{V}(P), \mathcal{V}(Q))$ is just $(\mathcal{V}(P) \leq \mathcal{V}(Q))$. For higher dimensions, other results are possible (e.g., a dot product called the *performance level* measure which compares alternative software designs [1]).

$\mathcal{T}$ is our decision process which determines how well $\mathcal{V}$ approximates our complexity ranking $\mathcal{R}$ between P and Q based on the measured complexity values $\mathcal{V}(P)$ and $\mathcal{V}(Q)$. We would like the relationship to be $\mathcal{T}(\mathcal{V}(P), \mathcal{V}(Q)) \Longleftrightarrow \mathcal{R}(P,Q)$, and in fact it is an implied axiom in most other complexity models. However, we believe that this is the major weakness that has prevented most complexity models from being truly effective. Because of undecidability issues (e.g. theorem T1), for all P and Q we cannot determine $\mathcal{T}$ for every $\mathcal{R}$. As a result, we use a weaker condition, namely:

Axiom A4: $(\forall P, Q)\ (\mathcal{R}(P,Q) \Rightarrow \mathcal{V}(P) \leq \mathcal{V}(Q))$

Since for many useful applications, $\mathcal{R}$ defines a total ranking, we then have:

Theorem T5: When $\mathcal{R}$ is total, i.e., $(\forall P,Q)\mathcal{C}(P,Q)$, we have:

$$(\forall P,Q)\ (\mathcal{V}(P) < \mathcal{V}(Q) \Rightarrow \mathcal{R}(P,Q))$$

In order to be useful, we would like our complexity measures to distribute programs across a range of values. If there is only a single "dominating" cluster point, we gain little information from the measure. Axiom **A5** allows, for rough comparisons, bi-polar or multi-polar distributions:

Axiom A5: $(\forall k \in \mathbb{R})(\exists \delta > 0)\ (|\mathrm{U} - \{P : \mathcal{V}(P) \in [k-\delta, k+\delta]\}| = |\mathrm{U}|)$

Axiom **A5** forces our complexity measure to be nontrivial, as in:

Theorem T7: $(\forall P)(\exists Q)\ (\mathcal{V}(P) \neq \mathcal{V}(Q))$

When $\mathcal{V}$ maps programs into a discrete bounded set S, axiom **A5** requires that at least two points in S have infinitely many programs with such values:

Theorem T8: If set S of complexity values is finite, then:

$$|\{k : (k \in \mathrm{S}) \wedge (|\{P : \mathcal{V}(P) = k\}| = |\mathrm{U}|)\}| \geq 2$$

3.2 A classification model

Given these five axioms, we developed a classification model for categorizing the various complexity measures depending upon the information they provide. A vertical classification uses a subset of the attributes for the entire program, while a hierarchical classification uses some functional relationship among the program's parts.

Vertical classification

A complexity ranking $\mathcal{R}$ is *abstract*, denoted $AB(\mathcal{R})$, if given P and Q with $AST(P) = AST(Q)$, then $\mathcal{R}(P,Q)$(and equivalently, $\mathcal{R}(Q,P)$).

If two programs are syntactically identical except for variable names, as long as two set of names are isomorphic, the only conceivable differences is interpretational (the meaning attached to each name). On the other hand, when considered functionally, each name is just a surrogate for the underlying data object. Thus we have the classification:

A complexity ranking $\mathcal{R}$ is *functional*, denoted $FN(\mathcal{R})$, if given P and Q with name sets α and β such that $AST(P^{\alpha}_{\beta}) = AST(Q)$, then $\mathcal{R}(P,Q)$.

Hierarchical classification

Assessing complexity by using only the components while ignoring interactions (i.e. ignoring the context where the components are defined and used) results in a *context free* ranking: A complexity ranking $\mathcal{R}$ is *context free*, denoted $CF(\mathcal{R})$, if given P, its ranking with respect to any given Q can be uniquely determined by: (1) Q and (2) $root(P)$, the complexity ranking of $left(P)$, and the complexity ranking of $right(P)$.

As a special case of context free complexity where organizational information is completely ignored, we can have primitive complexity: A complexity ranking $\mathcal{R}$ is *primitive*, denoted $PR(\mathcal{R})$, if all programs P and Q with the same collection of $AST(P)$ and $AST(Q)$ nodes (same number of occurrences for each corresponding pair) then $\mathcal{R}(P,Q)$.

Also, a complexity ranking $\mathcal{R}$ is *interactional*, denoted $IA(\mathcal{R})$, if it is not context free, i.e. $\neg CF(\mathcal{R})$.

Without considering interaction, the complexity of the composite complexity is the sum of all the components complexities. However, due to interaction among component parts, the total complexity may be greater than the sum. Such a complexity ranking is called *overall*.

If we are allowed to modify the internal structure, or reorganize the program according to some programming practices (such as modularization, data abstraction and information hiding), we may be able to cut down the interfacing complexity, thus the overall complexity. Since the two programs are functionally equivalent, they are comparable in complexity (**A2**).

The relationship among different hierarchical classes can be summarized in the following tree:

$$\textit{Hierarchical}\begin{cases}\begin{array}{c}\textit{Context Free}\\ CF(\mathcal{R})\end{array}\begin{cases}\textit{Primitive}\\ \quad PR(\mathcal{R})\\ \textit{Non Primitive}\\ \quad \neg PR(\mathcal{R})\end{cases}\\ \\ \begin{array}{c}\textit{Interactional}\\ IA(\mathcal{R})\end{array}\begin{cases}\textit{Overall}\\ \quad OA(\mathcal{R})\\ \textit{Not Overall}\\ \quad \neg OA(\mathcal{R})\end{cases}\end{cases}$$

Using this model, we have been able to develop Weyuker's 9 properties for complexity measures as special cases of our axioms [5]. Since those properties have been widely studied over the past 4 years, and since we can model her properties with our classification model, we believe that our axioms are a reasonable approximation of program complexity.

4 Application of the Model

Sixteen software systems, ranging from 3000 to 112,000 lines of FORTRAN source code, were selected from NASA ground support software for unmanned spacecraft control developed in the NASA/GSFC Software Engineering Laboratory. Each required between 5 and 140 person-months to develop over a period of 5 to 25 months by 4 to 23 persons. Each project contains from 83 to 531 *modules*, totalling over 4700 modules. There are 74 attributes, each quantified by a specific measure, for each module divided into three broad categories: fault, effort, and style (or complexity).

For each application instance, one of the projects was used as a training project in order to develop the classification tree for the next project. This was repeated for the remainder of the 16 projects.

Five of the projects were of a greatly different size than the others (by more than a factor of 3). We deemed these to not fulfill **Axiom A1'** on similarly of functionality. This reduced the set of projects to 11 (and 10 data points) and are given as Group A in what follows. We used a different ordering of 6 of the projects in terms of training set to give us Group B (and 5 additional data points). CTA refers to the original Classification Tree Analysis process, while ACT refers to the Axiomatic Classification Tree process developed in this paper.

4.1 Measure Screening

From the set of 74 measures for each module, we first eliminate all measures that are not directly measureable from the modules themselves. Thus effort data, e.g., number of hours to develop the module, are eliminated. We also eliminated change and error data since they represent interactions among program components and the operational environment. We can therefore reduce the number of measures to 40.

All candidate measures satisfy axioms **Axiom A1'** (comparing functionally equivalent programs), **Axiom A2** (comparing component-composite pairs), **Axiom A4** (measures agree with their ranking), and **Axiom A5** (no single cluster). However, many of the measures do not satisfy **Axiom A3**, the general monotonicity axiom. These measures are averaging measure such as ***assignment statements per 1000 executable statements***, which may be correlated with average effort per 1000 lines or so, but not with the total development effort. Therefore these measures will be eliminated. This reduces the candidate measures from 40 to 18, with the candidate measure set S being the left half of Table 2.

Both abstract and non-abstract aspects contribute to cost, so measures from any vertical class are potentially acceptable. On the other hand, as we are only considering cost and complexity at the module level, the hierarchical classification is not relavent. The analysis based on the measure classification scheme does not eliminate any measure for CTA in this case.

4.2 Aggregate Evaluation

Given 18 remaining measures that meet the boundary conditions based on the axioms and measure classifications, we next determine which of them best predicts total effort. The underline distribution, as we assumed, is a four region distribution (grouped into four quartiles) determined by historical data. A quartile of modules is positively identified if more than 75% of the modules (tolerance level: 25%) have the upper most quartile of effort. The negative sets can be similarly identified.

Let $m_i(\mathcal{V})$ $(i = 1,2,3,4)$ be the number of modules in quartile i using measure $\mathcal{V}$; $p_i(\mathcal{V})$ be the proportion of modules in $m_i(\mathcal{V})$ belonging or to the upper most quartile of effort; and $n_i(\mathcal{V})$ be the rest proportion in $m_i(\mathcal{V})$ (therefore $p_i(\mathcal{V}) + n_i(\mathcal{V}) = 1$). As a result, a quartile is positively identified if $p_i(\mathcal{V}) \geq 0.75$, and

Meets Axiom A3	Fails Axiom A3
	assignment statements per 1000 executable statements
	input-output statement per comment
assignment statements	input-output parameters per comment
input-output statements	input-output statements per 1000 executable statements
input-output parameters	input-output statements per input-output parameter
source lines	input-output statements per 1000 source lines
comments	function calls per comment
source lines minus comments	function calls per input-output statement
executable statements	function calls per function plus module call
function calls	function calls per input-output parameter
module calls	function calls per module call
function plus module calls	module calls per comment
cyclomatic complexity	module calls per input-output parameter
operators	module calls per function plus module call
operands	module calls per input-output statement
total operators	function plus module calls per 1000 source lines
total operands	function plus module calls per input-output statement
decisions statements	function plus module calls per input-output parameter
format statements	function plus module calls per 1000 executable statements
origin	function plus module calls per comment
	cyclomatic complexity per 1000 source lines
	cyclomatic complexity per 1000 executable statements

Table 2: Attributes passing initial screening

negatively identified if $n_i(\mathcal{V}) \geq 0.75$.

To formulate the objective function for the aggregated selection, we need to evaluate the contribution of each quartile. We can weight them by the number of modules falling into the quartile. Therefore, we formulate our selection criteria as:

$$\max_{\mathcal{V}, \mathcal{V} \in S} \left\{ \sum_{i}^{4} \{ m_i(\mathcal{V}) * p_i(\mathcal{V}) + m_i(\mathcal{V}) * n_i(\mathcal{V}) \} \right\} \qquad (1)$$

for i ranging from 1 to $p_i(\mathcal{V}) \geq 0.75 \vee n_i(\mathcal{V}) \geq 0.75$

This selection criterion maximizes the number of modules in positively or negatively identified quartiles. For each of the quartiles neither positively nor negatively identified, another measure is selected using the same criterion. The process continues until all modules are identified or all measures are exhausted.

5 Results

We applied both the original CTA process and the modified ACT process to the 16 NASA projects broken down into the 11 projects of groups A and six projects of B. The following sections describe the results of this analysis.

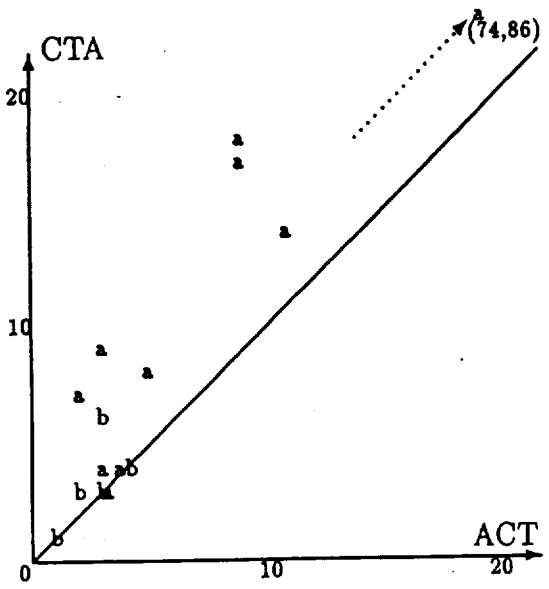
Size of generated trees

One measure of the efficiency of the technique is the size of the classification trees that are generated. Figure 2 shows that the axiomatic model (ACT) reduces tree size approximately 27% over the original CTA model from 188 nodes to 136 nodes in the 15 programs with average tree size dropping from 12.5 to 9.1 nodes.

The smaller the tree the more desirable (less costly to use to navigate through the tree, fewer measures to collect), thus a point in the upper left region represents an improvement over the original CTA.

Performance coverage

Table 3 compares the coverage based on the original and modified classification trees. In all the projects except one, near 100% coverage is achieved by both methods. Thus the decision tree analysis method almost always will predict a cost for a module and will



	individual data points		average		
	group A	group B	A	B	all
CTA	17 15 7 8 86 4 9 3 4 18	3 6 3 4 1	17.7	3.4	12.5
ACT	9 11 2 5 74 4 3 3 3 9	3 3 2 4 1	12.3	2.6	9.1

Figure 2: Internal Node Count Comparison

	group A	group B
CTA	98 98 99 98 91 93 97 100 100 98	100 98 97 97 100
ACT	99 100 97 100 82 93 100 98 100 99	98 98 100 97 100

a. individual data points

	group A	group B	all
CTA	97	99	97
ACT	97	99	97

b. average comparison

Table 3: Coverage Comparison

rarely leave modules unclassified. So, we can conclude that the CTA technique using either selection method achieves fairly good and consistent coverage, with an average of 97% coverage for both.

Performance accuracy

Accuracy improved about 5% with the ACT process, as given in Table 4.

Performance consistency

Table 5 gives the consistency comparison. This is the measure that drives the whole process, being that identification of high cost modules is the major goal

	individual data points		average		
	group A	group B	A	B	all
CTA	66 76 78 63 53 67 71 85 73 71	70 50 81 77 58	70	68	69
ACT	67 73 80 66 50 67 81 83 73 89	79 54 86 85 58	75	74	74

Table 4: Accuracy Comparison

	individual data points		average		
	group A	group B	A	B	all
CTA	70 66 31 54 52 63 30 16 50 10	7 100 33 17 65	39	35	38
ACT	67 61 37 57 56 63 50 15 50 23	43 85 40 29 65	50	50	50

Table 5: Consistency Comparison

	individual data points		average		
	group A	group B	A	B	all
CTA	26 60 54 62 42 21 42 33 6 47	7 4 40 63 47	38	28	35
ACT	30 46 73 59 49 21 14 33 6 30	71 13 40 63 47	35	39	35

Table 6: Completeness Comparison

of the prediction process.

The performance level between the two selection methods is significantly different, with the modified ACT selection method outperforming the original CTA method by a margin of 50% to 38%.

Performance completeness

While ACT generates many fewer "false alarms," (i.e., predicting high cost modules which really are not high cost – the above consistency measure), both methods are comparable in actually identifying the high cost modules, i.e., the completeness measure of Table 6. That is, both will fail to indicate high cost modules in over half the cases.

6 Conclusions

Classification Trees are a method to use measureable quantities from program modules in order to determine desireable attributes from the development process. Identification of high cost modules should correlate closely with other process measures such as reliability.

In this paper, we presented a Classification Tree Analysis (CTA) method and a modification to it, the Axiomatic Classification Tree Analysis (ACT) method, where an axiomatic model of program complexity was used to develop the candidate measures in the classification tree.

In all important measures, the ACT was either as good as or improved upon the original CTA model: (1) Classification trees were smaller; (2) Coverage was the same; (3) Accuracy improved; (4) Consistency improved and (5) Completeness was the same. We therefore believe that we have a candidate process that improves upon the original model.

Using an axiomatic basis for classification trees has two important economic benefits:

1. By eliminating unnecessary measures from the classificaiton tree (e.g., reducing the list from 74 to 18 in the NASA SEL experiment), we eliminate the need to collect such data. This would imply less overhead on the development process.
2. The axiomatic classification tree analysis technique generates improved results, allowing management to better control and evaluate the development process and allow for more informed decision making with less risk involved.

Of course there is still much more to be done. ACT is only right on 50% of the modules it calls high cost, and only finds accurately over one third of these modules. However, the method is improving, and is inexpensive to use since it is available as a byproduct of static analysis of the developing code. Further work will continue on developing these models.

Acknowledgements

This research was supported in part by National Science Foundation grant CCR-8819793 and National Aeronautics and Space Administration grant NSG-5123 to the University of Maryland.

References

[1] Cárdenas S. and M. V. Zelkowitz, "A management tool for the evaluation of software designs," *IEEE Trans. on Software Engineering* 17, 9 (September, 1991) 961-971.

[2] Porter A. A. and R. W. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees", *IEEE Software*, (March, 1990) 46-54.

[3] Selby R. W. and A. A. Porter, "Learning from example: Generation and evaluation of decision trees for software resource analysis," *IEEE Trans. on Software Engineering* 14, 12 (1990) 1743-1757.

[4] Tian J. and M. V. Zelkowitz, "A formal program complexity model and its application," *J. of Systems and Software* 17, 3 (March, 1992) 253-266.

[5] E. J. Weyuker, "Evaluating Software Complexity Measures," *IEEE Trans. on Software Engineering*, 14, 9 (1988) 1357-1365.