International Journal on Artificial Intelligence Tools
© World Scientific Publishing Company

Learning Satisficing Control Policies for Software Projects

Tim Menzies
Lane Department of Computer Science and Electrical Engineering
West Virginia University,
Morgantown, WV 26501
tim@menzies.us

James Kiper
Computer Science & Software Engineering Department
School of Engineering & Applied Science, Miami University
Oxford, OH 45056
kiperjd@muohio.edu

Jeremy Greenwald
Computer Science, Portland State University
jegreen@cecs.pdx.edu

Ying Hu
Software designer in Vancouver, British Columbia
huying_.@yahoo.com

David Raffo
School of Business Administration, Portland State University
raffod@sba.pdx.edu

Siri-on Setamanit
Portland State University
sirion@pdx.edu

Received (Day Month Year)
Revised (Day Month Year)
Accepted (Day Month Year)

Models in software engineering allow developers to define, but not necessarily explore, the space of design options. Building a useful model and understanding all its interactions, can be intellectually difficult - particularly for large, non-linear, discrete models. We show theoretically and empirically that our TAR2 *minimal contrast set* learner can generate very succinct conclusions from complex spaces.

Keywords: software process modeling, contrast set learning, treatment learning

1. Introduction

Anthropologists argue that the ability to build abstract models is what gives homo sapiens their competitive edge. In his article "What Models Mean" [63], Seidewitz describes the interactions and relationships among the concepts of a model. He asserts that "a model's meaning has two aspects: the model's relationship to what's being modeled and to other models derivable from it. Carefully considering both aspects can help us to understand how to use models to reason about the systems we build ..." For example, is common practice to build software process models to discover and exploit interesting interactions within a software project.

Software process models of modern systems are often large and complex. For example, many such models have an exceedingly large number of control parameters. The task of determining an optimal set of choice of these parameters is often humanly impossible. Experienced managers are able to make guesses that are sometimes adequate. However, it has become obvious that automated help for managers of large system developments is vital.

In this work, we will describe a tool (TAR2) that, for many hard models, can identify a small number of controlling variable that are the keys to a model's function. We call this set of controlling variables a *collar.* (This term will be explicitly defined in a later section.) Such succinct controllers have many advantages:

- Smaller models are easier to understand and explain (or audit).
- Miller has shown that models generally containing fewer variables have less variance in their outputs [48].
- The smaller the controller, the fewer are the demands on interfaces (sensors and actuators) to the external environment. Hence, controllers for softare processes designed around small models are easier to use (i.e., fewer things to do) and less expensive to build.

Previously, we have explored the generation of such succinct controllers with TAR2, in the context of small models with regular topologies. For example, TAR2 has beeen applied to

(1) COCOMO models to create designs with lower effort and fewer risks [47] and defects [45].
(2) The NASA SILAP model for selecting V&V tasks [23].
(3) Finite state machines to find topologies that reduce the CPU cost of applying formal methods [44, 52].
(4) Maximizing whiskey production, modeled as finite state machines [9];

The above results, while interesting, are not convincing evidence of the scalability of TAR2. The COCOMO and SILAP models are not large (the COCOMO model consists of four equations and the SILAP model is a single-parent tree, 3 layers deep). As to the other examples, they have a regular and repeated topology: in the above examples, the same network of machines occurs in mutliple places around the model. Such repeated sub-structures can greatly simplify any search over their space of internal options.

Therefore, in this paper, we turn our attention to bigger and more complex models. The two case studies presented below describe models that have been built and patched over many years:

- A model of inspection policy effectiveness for a large software house;
- The Software Engineering Institute's capability maturity model (level 2);

These are large models: unlike the COCOMO and SILAP models described previously, their specification fills many pages. Also, unlike our finite state machine models, the internal structure of these models does not repeat sub-structures. Hence, they are a more challenging problem for TAR2.

This paper tests TAR2 on these more complex models. First, we discuss modeling in software engineering and highlight the repeated observation that even seemingly simple models can contain unexpected interactions. As model complexity grows, it is therefore vital to augment model creation with automatic tools that search for unexpected or detremential interactions. Next, we then discuss "hard modeling" problems. When making descisions in limited time using limited information about the environment, it is useful to know the *fewest* decisions that *most affect* the outcomes. One way to find these smallest decision sets is to exploit a phenomenum that we call "collars" and "clumps". These are features of search space that have been reported many times previously but, we believe have not been fully exploited. Finally, this paper discusses how *collars* and *clumps* changes the design of data miners. Our tool that exploits collars and clumps (TAR2) is then applied to two case studies.

As we shall see, despite uncertainties or variabilities in the model, TAR2 was able to find inputs that led tor preferred model output. Hence, these experiments increase our confidence that TAR2 is a valuable addition to augment standard modeling methods.

2. Models Hide Errors

Model are useful since humans can review/audit/improve an explicit representation of their systems in a more effective and efficient manner before actually implementing that system. But manual inspection is often insufficient to reveal the subtleties of a model. Even very simple models can hide a surprising number of errors. Consider, for example, a simple mathematical model of population growth.

$$\frac{dN}{dT} = rN \tag{1}$$

Here, r is a constant reflecting environmental conditions, T is time, and N is the population. Note that this model is not accurate since population growth must taper off as it approaches c, the maximum carrying capacity of the environment; i.e.

$$\frac{dN}{dT} = rN\left(1 - \frac{N}{c}\right) \tag{2}$$

Before reading further, we ask the reader to consider whether equation #2 is an appropriate model of population growth? If the reader cannot see all the subtleties in a one-line model, then we should be suspicious of claims that the validity of larger models can be accurately determined by manual analysis. Although equation #2 does model our intuition in some cases, there is one situation in which it is clearly incorrect. Consider the case of over-population in an hostile environment: $N > c, r < 0$. Our intuition is that, in that situation, the population will fall. However, with these assumptions $rN(1-(N/c)) > 0$. That is, our model concludes that the population will *increase* (example taken from [37].)

Our experience has been that this error in this simple model is not apparent to many people. Myers [50] reports a similar conclusion, but using a 63 line model. In this experiment, 59 experienced IT professionals searched for errors in a very simple text formatter that consisted of 63 lines of PL/1 code. Even with unlimited time and the use of three different methods, 73% of the experts could only find (on average) 5 of the 15 errors in this 63 line model [50]. This result, despite its age, and the previous thought experiment do not inspire confidence that experts can accurately assess larger models.

This phenomenon of models hiding errors is not limited to software. Consider the results of the Feldman and Compton study in which 109 of 343 (32%) of the known data points from six studied papers could not be explained using a glucose regulation model developed from international refereed publications [20, 21, 65]. A subsequent study corrected some modeling errors of Feldman and Compton to increase the inexplicable percentage from 32% to 45%. A similar study successfully faulted another smaller published scientific theory [43].

But as models grow in complexity, it becomes difficult for a manual analysis to reveal all their subtleties. Hence, many researchers propose support environments to help explore the increasingly complex models that engineers are developing. Gray, et al, [24] have developed the Constraint-Specification Aspect Weaver (C-Saw) that uses aspect-oriented approaches [22] to help engineers in the process of model transformation. Cai and Sullivan [11] describe a formal method and tool called *Simon* that "supports interactive construction of formal models, derives and displays design structure matrices ... and supports simple design impact analysis." Other tools of note are lightweight formal methods such as ALLOY [32] and SCR [28] as well as various UML tools that allow for the execution of life cycle specifications (e.g. the CADENA scenario editor [12]).

Recently, artificial intelligence (AI) methods have been successfully applied to model-based SE. For example, Whittle uses deductive learners to generate lower-level UML designs (state charts) from higher-level constructs (use case diagrams) [69]. More generally, the field of *search-based SE* augments model-based SE with *meta-heuristic* techniques, like genetic algorithms, simulated annealing, etc., to explore a model. Such heuristic methods are hardly complete but, as Clarke, et al [13] remark: "...software engineers face problems which consist, not in finding *the* solution, but rather, in engineering an *acceptable* or *near optimal solution* from a large number of alternatives." [13].

Search-based SE is most often used to optimizing software testing [33,34,53,66] but it has had application in numerous other areas. In prior work with Martin Feather [18], we have used search-based SE for requirements analysis. Other researchers [25, 38] use genetic algorithms to examine ways of modularizing software [13] or developing effort estimators [2,14,15]. In all, Rela [59] lists 123 publications where search-based methods have been applied to the above applications as well as automatic synthesis of software defect predictors, assisting in component design, developing multiprocessor schedules, re-engineering old systems into a better one, and searching for compiler optimizations.

To use a search-based approach, software engineers have to reformulate their problem by:

- Finding a *representation of the problem* that can be symbolically manipulated (e.g. simulated or mutated). Such representations always exist with model-based SE.
- Defining a *fitness function* (a.k.a. "utility function"" or "objective function"); i.e. an "oracle" that scores a model configuration.
- Determining an appropriate set of *manipulation operators* to select future searches based on the prior searches [26].

The rest of this paper explores data mining as one way to implement automatic manipulation operators. Our TAR2 data miner searches through the space of possible concepts for a combination of concepts that describes some target theory. Given, say, the output from a Monte Carlo simulation of a model, TAR2 can sift through model output looking for the core concepts that most often led to preferred output.

3. "Hard Modeling" Problems

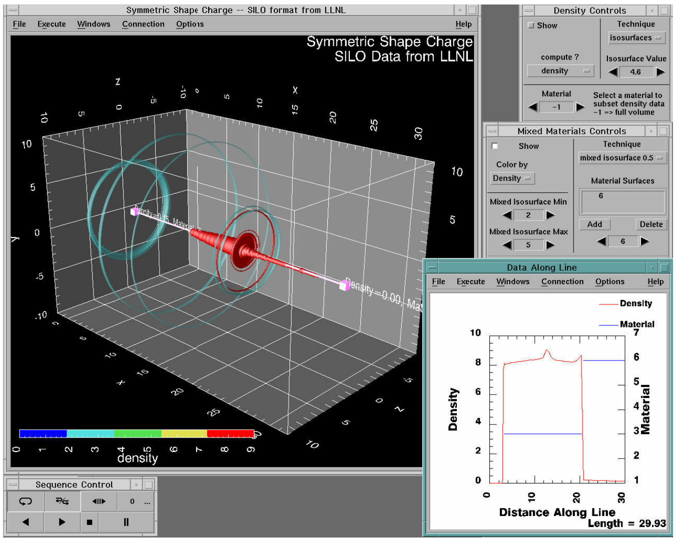
Before describing our research into data mining for automatically generating manipulation operators, we must motivate why we do not use traditional methods. We assert that may models of large software projects follow into the category of' "hard modeling" problems. These are models in which an optimal answer is not possible, uncertainty permeates, the behavior is non-linear, the size and complexity compound to make the model difficult to understand. Thus, we seek constraints to model inputs that are *satisficing* (rather than optimal), are stable in the face of uncertainty, can be automatically generated, reduce cognitive overload, and which work for large non-linear and noisy models. Such constraints are called "solutions to hard modeling problems".

Many researchers have developed impressive visual environments for decreasing the cognitive overload associated with exploring a multi-dimensional space. For example, Figure 1 shows a tool developed at IBM that augments a standard three-dimensional display with visual cues relating to higher-dimensional data; e.g., supplementary dimensions are shown bottom right; blue circles around the axle show circular motion information; and a color key, shown bottom left, indicates how colors in the display relate to density information [67].

Such visualizations help analysts explore visual information, but they present their own challenges. There are still limits on how many dimensions can be displayed (e.g. we have yet to see effective visualizations for more than a ten dimensionality space). Also, note how the tool shown in Figure 1 contains numerous controls that allow analysts to change various visualization options. As the visualization environment grows more sophisticated, some users find they have traded a data browsing problem with the new problem of exploring the full range of the effects of all the controls.

When manual exploring of options fail, automatic methods can be applied. Optimization packages can be applied to data or the equations of a system to find "sweet spots" that maximize the score resulting from model outputs. Related methods include *sensitivity analysis* [61] and *design of experiments* (DOE) methods [7] A canonical sensitivity analysis method might be to compute eigenvectors of a linear system in order to understand its long-term temporal behavior [31, 37]. As for design of experiments, DOE exercises an existing model and helps shed light on the response surface of the model. DOE does identify gradients and key parameters for a model.

While useful for some models, these automatic methods do not apply to all



Symmetric Shape Charge -- SILO format from LLNL

Symmetric Shape Charge
SILO Data from LLNL

Density Controls

□ Show

Technique: isosurfaces

compute ? density

Isosurface Value: 4.6

Material: -1

Select a material to subset density data
-1 => full volume

Mixed Materials Controls

□ Show

Technique: mixed isosurface 0.5

Color by: Density

Material Surfaces: 6

Mixed Isosurface Min: 2

Mixed Isosurface Max: 5

Data Along Line

Length = 29.93

Sequence Control

Fig. 1. A visualization tool for scientific data. From [67].

models. For example, optimization methods can fail for non-linear models. Any model with an "if" statement potentially introduces a "cliff" where the effects of inputs on outputs abruptly change. For such models, there is no linear continuous solution that applies to both sides of the "cliff". Model uncertainty also complicates sensitivity analysis. For example, the eigenvector technique described previously would yield spurious results if the coefficients on the models are not known with certainty. Lastly, a DOE analysis can be complicated by the dimensionality, noise, and visualization problems described above.

In *hard modeling problems*, human agents must make decisions using:

- limited time;
- limited computational ability (or limited time for computation);
- limited knowledge about decision alternatives;
- uncertainty about possible outcomes of decisions;
- no more than a partial ordering of preferences;
- limited information about probabilities of outcomes.

Herbert Simon [64] defined and explored such hard modeling problems using a data structure called *state space* [60]. In terms of model-based software engineering (SE), a state space is the set of options and option selection operators within a model. In hard modeling problems large portions of the state space are uncertain. Simon argued that in such state spaces, searching for optimal solutions is a spurious goal. Rather, agents can only make just enough decisions that are just good enough. In Simon's terminology, such decisions are *satisficing*.

Our contribution to hard modeling is to comment that (1) *satisficing* solutions often can be achieved by ignoring certain irrelevant or redundant details within a model; (2) surprisingly simple methods often can find what details are relevant and what can be ignored. TAR2 exploits these properties of hard models through a form of data mining called /em treatment learning. A treatment learner proposes "treatments"; i.e., constraints on a small subset of the model inputs. The other inputs are left to vary at random. (The term "treatment" is chosen as an analogy to the health professions use of that term in which a set of activities, medication, and/or procedures is prescribed to address some specific health issue.) The result is that, in addition to generating very succinct solutions, treatment learning offers solutions that are stable despite uncertainty in the non-treated variables.

Our own view on hard modeling is that there often exists a "loophole" in search problems that can make hard problems far easier to manage. We call this loophole "collars and clumps".

4. Collars and Clumps

This research assumes that many models can be controlled by a small number of key variables which we call *collars*. Collars restrict the behavior of a model such that their state space *clumps*; i.e. only small number of states are used at runtime.

The output of a data miner can be simplified to constrain just the collar variables that switch the system between a few clumps.

Definition: A *collar* is a set of input variables that determines the state of a system in most cases. The *size* of the collar is the ratio of the cardinality of this collar set to the total number of input variables for the system.

Definition: A dynamic model exhibits *clumps* when most values of input variables result in the systems being in one of relatively few states.

From these definition, it is apparent that these two concepts are duals. A system with an effective collar is one that exhibits a high degree of clumping, and vice versa. To visualize collars, imagine an execution trace spreading out across a program. Initially, the trace is very small and includes only the inputs. Later, the trace spreads wider as *upstream* variables affect more of the *downstream* variables (and the inputs are the most *upstream* variables of all). At some time after the appearance of the inputs, the variables hold a set of values. Some of these values were derived from the inputs while others may be default settings that, as yet, have not been affected by the inputs. The union of those values at time t is called the *state* s_t of the program at that time.

Multiple execution traces are generated when the program is run multiple times with different inputs. These traces reach different branches of the program. Those branches are selected by tests on conditionals at the root of each branch. The *controllers* of a program are the variables that have different settings at the roots of various branches in separate traces. Programs have $collars$ when a handful of the controllers in an early state s_t control the settings of the majority of the variables seen in later states.

As described previously *collars* are related to *clumping.* If a program has v variables with range r, then the maximum number of states is r^v. Programs *clump* when most of those states are never *used* at runtime; i.e. $|used|/\left(r^v\right) \approx 0$. Clumps can cause collars:

- The size of *used* is the cardinality of the cross product of the ranges seen in the controllers.
- If that cardinality is large, many states will be generated and programs do not clump.
- But if that cross product cardinality is small, then the deltas between the states will be small – in which case controlling a small number of collar variables would suffice for selecting which states are reached at runtime.

As shown below, there is much theoretical and empirical evidence for expecting that many models often contain *collars* and *clumps.*

4.1. *Theoretical Evidence*

In prior work with Singh [46], we have shown that collars are an expected property of Monte Carlo simulations where the output has been discretized into a small number of output classes. After such a discretization, many of the inputs would reach the same goal state, albeit by different routes. The following diagram shows two possible distinct execution paths within a Monte Carlo simulation both leading to the same goal; i.e. $a \rightarrow goal$ or $b \rightarrow goal$.

$$\left.\begin{array}{r}\xrightarrow{a_1} M_1\\ \xrightarrow{a_2} M_2\\ \dots\\ \xrightarrow{a_m} M_m\end{array}\right\} \xrightarrow{c} goal_i \xleftarrow{d} \left\{\begin{array}{l} N_1 \xleftarrow{b_1}\\ N_2 \xleftarrow{b_2}\\ N_3 \xleftarrow{b_2}\\ N_4 \xleftarrow{b_2}\\ \qquad\dots\\ N_n \xleftarrow{b_n}\end{array}\right.$$

Each of the terms in lower case in the above diagram represent a probability of some event; i.e. $0 \leq \{a_i, b_i, c, d, goal_i\} \leq 1$. For the two pathways to reach the *goal*, they must satisfy the collar M or the larger collar N (each collar is a conjunction). As the size of N grows, the product $\prod_{j=1}^{N} b_j$ decreases and it becomes less likely that a random Monte Carlo simulation will take steps of the larger collar N.

The magnitude of this effect is quite remarkable. We have executed the above model under a variety of conditions:

- A basic simulation, where we assume where $b_i = b_j = \frac{1}{n}$ and $a_i = a_j = \frac{1}{m}$;
- A more complex simulation where a_i and b_i are drawn from random distribitions. The complex simualtion results were described in [46]. We do not repeat those results here since they report the same pattern as seen in the basic simulation.

According to the basic simulation, the narrower collar is thousands to millions of times more likely. For example, when $|M| = 2$ and $N > M$, the condition for selecting the larger collar is $\frac{d}{c} \geq 64$; i.e. the larger collar N will be used only when the d pathway is dozens of times more likely than c. The effect is more pronounced as $|M|$ grows; at $|M| = 3$ and $N > M$, the condition is $\frac{d}{c} \geq 1728$; i.e. to select the larger collar N, the d pathway must be thousands of times more likely than c (for more details, see [46]). That is, when the output space is discretized into a small number of outputs, and there are multiple ways to get to the same output, then a randomized simulation (e.g. a Monte Carlo simulation) will naturally select for small collars.

As to *clumping*, Druzdel [16] observed this effect in a medical monitoring system. The system had 525,312 possible internal states. However, at runtime, very few were ever reached. In fact, the system remained in one state 52% of the time, and a mere 49 states were used, 91% percent of the time. Druzdel showed mathematically that there is nothing unusual about his application. If a model has n variables, each with its own assignment probability distribution of p_i, then the probability that the model will fall into a particular state is $p = p_1 p_2 p_3 ... p_n = \prod_{i=1}^{n} p_i$. By taking logs of both sides, this equation becomes

$$\ln p = \ln \prod_{i=1}^{n} p_i = \sum_{i=1}^{n} \ln p_i \tag{3}$$

The asymptotic behavior of such a sum of random variables is addressed by the central limit theorem. In the case where we know very little about a model, we assume that the p_i are uniform;y distributed and that many states are possible. However, the *more* we know about the model, the *less* likely it is that the distributions are uniform. Given enough variance in the individual priors and conditional probabilities or p_i, the expected case is that the frequency with which we reach states will exhibit a log-normal distribution; i.e. a small fraction of states can be expected to cover a large portion of the total probability space; and the remaining states have practically negligible probability.

The assertion that many types of models display this clumping behavior is quite important for the style of data mining (treatment learning) that we advocate. In application to a clumping model with collars, Monte Carlo simulation, followed by the use of TAR2, suffices to summarize that model in an effective way:

- TAR2's rules never need to be bigger than the collars. Hence, if the collars are small, TAR2's rules can also be small.
- If a model clumps, then, very quickly, a Monte Carlo simulation would sample most of the reachable states. TAR2's summarization of that simulation would then include most of the important details of a model.

4.2. *Empirical Evidence*

Empirical evidence for clumps first appeared in the 1950s. Writing in 1959, Samuel studied machine learning for the game of checkers [62]. At the heart of his program was a 32-term polynomial that scored different configurations. For example, *king center control* means that a king occupies one of the center positions. The program learned weights for these variable coefficients. After 42 games, the program had learned that 12 variables were important, although only 5 of these were of any real significance.

Decades later, we can assert that deleting irrelevant variable has proven to be a useful strategy in many domains. For example, Kohavi and John report experiments on 8 real world datasets where, on average, 81% of the non-collar variables can be ignored without degrading the performance of a model automatically learned from the data [35].

If models contain collars, or if the internal state space clumps, then much of the reachable parts of a program can be reached very quickly[a]. This *early coverage*

[a] Note that this is different to the *reliability* issue which is "which parts of the system, that we have *not* reached before, might we reach now." While our tool sumamrizes the key points in observed behavior, it can be used to drive the system into regions it does not frequently visit. For more on this issue, see [52]

effect has been observed many times. In a telecommunications application, Avritzer, Ros, and Weyuker found that a sample of 6% of all inputs to this system covered 99% of all inputs seen in about one year of operation (and a sample of just over 12% covered 99.9%) [3]. Further evidence for early coverage can be found in the mutation testing literature. In mutation testing, some part of a program is replaced with a syntactically valid, but randomly selected, variant (e.g. switching “less than” signs to “greater than”). This method of testing is useful for getting an estimate of what percentage of errors have been discovered by testing. Wong compared results using X% of a library of mutators, randomly selected (X $\in$\{10,15,. . . 40,100\}). Most of what could be learned from the program could be learned using only X=10% of the mutators; i.e. after a very small number of mutators, new mutators acted in the same manner as previously used mutators [70]. The same observation has been made elsewhere by Budd [8] and Acree [1].

If the space of possible execution pathways within a program is limited, then program execution would be observed to clump since it could only ever repeat a few simple patterns. Empirically such limitations have been observed in procedural and declarative systems. Bieman and Schultz [5] report that 10 or fewer paths through programs explored 83% of the du-pathways. (A *du-path* is a set of statements in a computer program from a definition to a use of a variable. This is one common form of structural coverage testing.) Harrold [27] studied the control graphs of 4000 Fortran routines and 3147 C functions. Potentially, the size of a control graph may grow as the square of the number of statements (in the case where every statement was linked to every other statement.) This research found that, in these case studies, the size of the control graph is a linear function of the number of statements. In an analogous result, Pelánek reviewed the structures of dozens of formal models and concluded that the internal structure of those models was remarkably simple: “state spaces are usually sparse, without hubs, with one large SCC [strongly connected component], with small diameter [b] and small SCC quotient'[c]” [55]. This sparseness of state spaces was observed previously by Holtzmann where he estimate the average degree of a vertex in a state space to be 2 [29].

Pelánek hypotheses that these “observed properties of state spaces are not the result of the way state spaces are generated nor of some features of specification languages but rather of the way humans design/model systems” [55]. Pelánek does not expand on this, but we assert that generally SE models are simple enough to be controlled by treatment learning since they were written by humans with limited short-term memories [49] who have difficulty designing overly-complex models.

[b] The diameter of a graph (of a state space here) is the number of edges on the largest shortest path between any two vertices.

[c] SCC quotient is a measure of the complexity of a graph.

5. Data Mining with Collars and Clumps using Treatment Learning

The TAR2 *treatment learner* [39, 41] is a data miner designed specifically to explore models collars. TAR2 finds the difference between outcomes. Formally, the algorithm is a *contrast set learner* [4, 68] that uses *weighted classes* [10] to steer the inference towards the preferred behavior. We call TAR2's output "treatments" since the minimal rules generated by the algorithm are similar to medical treatment policies that try to achieve the most benefit, with the least intervention. The core intuition of TAR2 is that it is unnecessary to search for the collars – they will reveal themselves after some limited random sampling. To see that, recall that collar variables control the settings in the rest of the system. Any execution trace that reaches a goal must pass through the collars (by definition). Therefore, to find the collars, all an algorithm needs to do is find the attribute ranges with very different frequencies in traces that reach different goals.

Detecting collars via this sampling method is quite simple to implement. Consider a log of golf playing behavior shown in Figure 3. This log contains four attributes (outlook, temperature, humidity, wind) and 3 target classes (none, some, lots) that convey the amount of golf played. We recommend an exponential scoring system for the classes, starting at two[d]. For example, our golfer could weight the classes in Figure 3 as *none=2* (worst), *some=4*, *lots=8* (best).

TAR2 seeks attribute ranges that occur frequently in the highly weighted classes and rarely in the lower weighted classes. Let $a.r$ be some attribute range, e.g. *outlook.overcast* means that the outlook is for overcast skies. $\Delta_{a.r}$ is a heuristic measure of the worth of $a.r$ to improve the frequency of the *best* class. $\Delta_{a.r}$ uses the following definitions:

$X(a.r)$: is the number of occurrences of that attribute range in class X; e.g. in this data *lots(outlook.sunny)=2* since there are 2 cases with outlook = *sunny* and class = *lots*.

$all(a.r)$: is the total number of occurrences of that attribute range in all classes; e.g. *all(outlook.sunny)=5.*

$best$: the highest scoring class; e.g. $best = lots$.

$rest$: the set of non-best class; e.g. $rest = \{none, some\}$.

$weight$: The weight of a class X is symbolized by $\$X$. (Thus, $\$best = 8$.)

$\Delta_{a.r}$ is calculated as follows:

$$\Delta_{a.r} = \frac{\sum_{X \in rest} (\$best - \$X) * (best(a.r) - X(a.r))}{all(a.r)} \tag{4}$$

[d] If the weights run, say, {bad=0,ok=1,good=2} then the difference from *bad* to *ok* scores the same as *ok* to *good*. An exponentially weighting scheme, starting at two, finds greater and greater rewards moving to better classes. For further details, see [30].

When $a.r$ is *outlook.overcast*, then $\Delta_{outlook.overcast}$ is calculated as follows:

$$\frac{\overbrace{((8-2)*(4-0))}^{lots \rightarrow none}+\overbrace{((8-4)*(4-0))}^{lots \rightarrow some}}{4+0+0}=\frac{40}{4}=10$$

In the following, we will refer to the Δ and Δs functions. The difference between these is that the Δ is a heuristic measure of a single attribute range while Δs is a heuristic measure of a conjuction of attribute ranges. (Thus, if the conjuction has a cardinality of one, then Δ=Δs). The Δs function is almost identitcal to Equation 4, but all references to $a.r$ are replaced by the frequency count of the number of rows selected by $(a1.r1 \wedge a2.r2...)$.

To *build* a treatment, TAR2 explores combinations of attribute ranges up to some user-specified maximum size s (where the size s is the number of attribute ranges in a conjunction of attributes). Given n attributes, the size of this search is $\frac{n!}{s!(n-s)!}$. To make this search feasible, TAR2 must keep s small. Therefore, TAR2 first assesses each attribute range, in isolation, i.e., with s = 1. A preliminary pass builds one singleton treatment for each attribute range. The attribute ranges are then scored by the Δ of these singleton treatments. Treatment generation is constrained to just the attribute ranges with a score greater than a user-supplied threshold.

```
input:  D          The example data.
        items      Attributes seen in the examples.
        s          Desired size of rule. Default=4.
        promising  Threshold for a useful attribute range. Detault= 1.5
        skew       Threshold for acceptable number of best entries in
                   treated. Default=20%
        bands      Number of divisions within continuous ranges. De-
                   fault=5.

output:            A conjunction of attribute ranges

01. $D_1 \leftarrow$ discretize($D$, $bands$)
02. $temp \leftarrow$ -1
03. for $attribute$ in $items$ {
04.     for $R$ in $attribute.ranges$ {
05.         if   $\Delta(attribute.R) \geq promising$
06.         then $candidates \leftarrow candidates$ + $attribute.R$}}
07. for $C \subseteq candidates$ where $|C| \leq s$ {
08.     $treated \leftarrow C \wedge D_1$
09.     $result \leftarrow \Delta$s($C$)
10.     if   $result > temp$ and $|best \wedge D_1|/|best \wedge treated| > skew$
11.     then   { output $\leftarrow C$
12.              $temp \leftarrow result$}
13. return output
```

Fig. 2. The TAR2 algorithm.

The TAR2 algorithm is shown in Figure 2:

- The `discretize` function on line 1 divides the numeric ranges seen in the examples into *bands* number of groups. TAR2 was originally designed using a very simple discretization policy; i.e. TAR2 sorts the known values and divides into *bands*with (roughly) the same cardinality. It was anticipated that this policy

outlook	*temp(^oF)*	*humidity*	*windy?*	*class*	*weight*
sunny	*85*	*86*	*false*	*none*	*2*
sunny	*80*	*90*	*true*	*none*	*2*
sunny	*72*	*95*	*false*	*none*	*2*
rain	*65*	*70*	*true*	*none*	*2*
rain	*71*	*96*	*true*	*none*	*2*
rain	*70*	*96*	*false*	*some*	*4*
rain	*68*	*80*	*false*	*some*	*4*
rain	*75*	*80*	*false*	*some*	*4*
sunny	*69*	*70*	*false*	*lots*	*8*
sunny	*75*	*70*	*true*	*lots*	*8*
overcast	*83*	*88*	*false*	*lots*	*8*
overcast	*64*	*65*	*true*	*lots*	*8*
overcast	*72*	*90*	*true*	*lots*	*8*
overcast	*81*	*75*	*false*	*lots*	*8*

Fig. 3. A log of some golf-playing behavior. From [57].

would be too simplistic and would have to be improved. However, our empirical results (see below) were so encouraging that we were never motivated to do so.

- Lines 3 to 6 show the preliminary pass to find *promising* attribute ranges.
- The rest of the algorithm tries all subsets of the promising ranges looking for the one that generates the largers Δs value. Line 10 checks for overfitting: if a treatment selects too few of the best classes in the data, it is ignored.

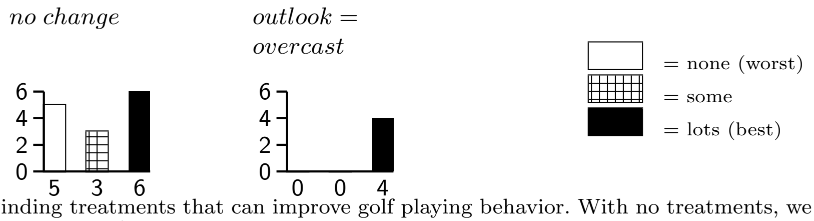


Fig. 4. Finding treatments that can improve golf playing behavior. With no treatments, we only play lots of golf in $\frac{6}{5+3+6}$ = 57% of cases. However, assuming *outlook=overcast*, we play golf lots of times in 100% of cases.

To *apply* a treatment, TAR2 rejects all example entries that contradict the conjunction of the attribute ranges in the treatment. E.g., if the treatment was $humidity \geq 85 \wedge windy = true$, then 11 of the lines of Figure 3 would be rejected. The ratio of classes in the remaining examples is compared to the ratio of classes in the original example set (in the humidity and wind treatment just given, this ratio would be 3/14). The *best treatment* is the one that most increases the relative percentage of preferred classes. In our golf example, a single best treatment was generated containing *outlook=overcast*. Figure 4 shows the class distribution before and after that treatment. That is, if we select a vacation location with *overcast* weather, then we should be playing *lots* of golf, all the time.

		# columns			
domain	# rows	# numeric	# discrete	#class	time(sec)
iris	150	4	0	3	< 1
wine	178	13	0	3	< 1
car	1,728	0	6	4	< 1
autompg	398	6	1	4	1
housing	506	13	0	4	1
pageblocks	5,473	10	0	5	2
cocomo	30,000	0	23	4	2
reacheness	25,000	4	9	4	3
circuit	35,228	0	18	10	4
reacheness2	250,000	4	9	4	23
pilot	30,000	0	99	9	86

Fig. 5. Runtimes for TAR2 on different domains. The first 6 data sets come from the UC Irvine machine learning data repository [6]; "cocomo" comes from a COCOMO software cost estimation model [47]; "pilot" comes from the NASA Jet Propulsion Laboratory [19]; "Reachness" and "Reachness2" come from [51]; "circuit" comes from [40].

In practice, despite the $\frac{n!}{s!(n-s)!}$ search, TAR2 scales well. Figure 5 shows TAR2's runtime on 11 data sets with varying numbers of rows and columns. Running on a relatively slow machine (a 333 MHz Windows machine with 512MB of ram), TAR2 terminated in tens of seconds, even on data sets with up to 250,000 rows, each with nearly 100 attributes.

6. Case Studies

In theory, we expect that many models contain collars and clumps. If so, tools like TAR2 should be able to find tiny treatments that control the behavior of the models. This section tests that theory on several case studies.

6.1. *Inspection Policies*

The first case study contrasts treatment learning with traditional learners. It will be seen that treatments are dramatically smaller, and more understandable, than the model learned by standard data miners.

Figure 6 offers a high-level view of a quantitative software process model [58]. At each phase of that process, inspections are conducted of the functional specification (FS), high-level design (HLD), low-level design (LLD), and the code (CODE). Raffo modeled these phases, and the inspections using a Statemate^{TM} state-based simulation model and an Extend^{TM} discrete event model containing 30+ process steps with two levels of hierarchy. Some of the inputs to the simulation model included productivity rates for various processes, the volume of work (i.e. KSLOC), defect

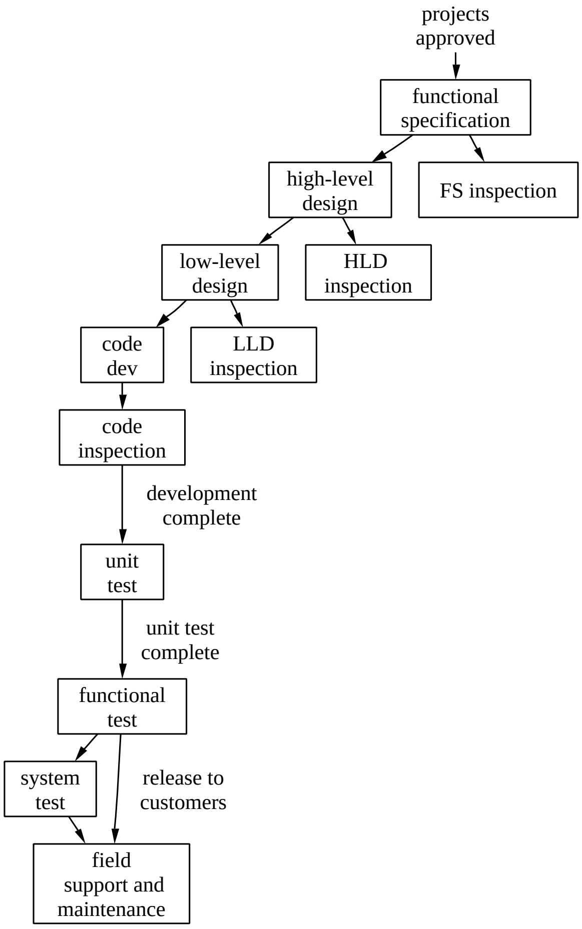


Fig. 6. High-level block diagram of a discrete event model of one company's software process.

detection and injection rates for all phases, effort allocation percentages across all phases of the project, rework costs across all phases, parameters for process overlap, the amount and effect of training provided, and resource constraints.

Model outputs are the development *expense* (person months), product *quality* (number of high severity defects) and project *duration* (calendar months) which are combined as follows:

$$\begin{aligned} utility = {} & 40 * (14 - quality) \; + \; 320 * (70 - expense) \, + \\ & 640 * (24 - duration) \end{aligned} \tag{5}$$

The justification for this style of utility function is discussed in detail in [58]. In summary, this function was created after extensive debriefing of the business users.

The model has been extensively validated. The model's process diagrams, model inputs, model parameters and outputs were reviewed by members of the software engineering process group as well as senior developers and managers. In other studies, the model was used to accurately predict the performance of several past releases of the project. Finally, in *special case* studies, the model was used to predict unanticipated special cases. Specifically, when predicting the impact of developing overly complex functionality, the model predicted that development would take approximately double the normal development schedule. Initially rejected by management, it was later found that this model's predictions corresponded quite accurately with the company's actual experience.

In this example, we will use the model to assess different software inspection policies. For each phase of the software process modeled in Figure 6, four types of inspections can potentially be applied. These four types are listed below, sorted by their cost and effectiveness. For example, *full Fagan inspections* are most expensive and find the most issues. At the other end of the scale, doing *no inspections* is cheapest but finds no issues:

F: A *full Fagan inspection* [17] is a seven step process with pre-determined roles for inspection participants. For the company studied by Raffo, the *defect detection capability*[e] of their full Fagan inspections was $TR(0.35, 0.50, 0.65)$[f]. Such studies use between 4 and 6 staff, plus the author of the artifact being inspected.

B: A *baseline inspection* is a continuation of current practice at the company under study. The baseline inspection at this company was essentially a poorly performed Fagan inspection, Historical records show that these baseline inspections have varying defect detection capabilities of $\{min, median, max\} = \{0.13, 0.21, 0.30\}$.

W: *Walk through* inspections conducted informally by an outside consultant. Historical records show that these inspections have a defect detection capability of $TR(0.07, 0.15, 0.23)$.

N: *No inspection*;

Each type of inspection can be performed at each phase; i.e. there are 4^4 possible inspection polices. A data set for TAR2 was prepared by running each configuration 50 times; i.e. $50 * 4^4 = 12800$ samples. Each run was then tagged with its utility, using Equation 5. These utilities were discretized into four classes, of approximately equal frequency. (Thus, the significance of the boundary values is just that they give class of approximately equal cardinality.)

- *class1* : value of Equation 5 < 9843
- *class2* : $9843 \leq$ value of Equation 5 < 10698
- *class3* : $10698 \leq$ value of Equation 5 < 11664
- *class4* : $11664 \leq$ value of Equation 5 ≤ 14755

[e] Defect detection capability is the percentage of defects detected from those that are latent in the inspected artifact.

[f] $TR(a, b, c)$ denotes a triangular distribution with minimum, median, max of a, b, c respectively.

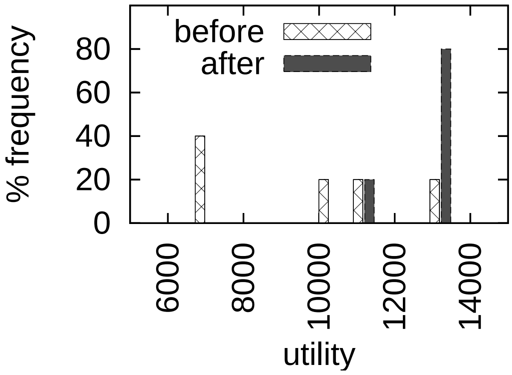


Fig. 7. Utility frequency in 12800 samples, *before* and *after* treatment.

The $before$ bar chart in Figure 7 shows the frequency of these classes in the untreated model. Note that many (40%) of the samples generate the lowest $class1$ utility.

TAR2 was then applied to learn treatments that distinguish the desired $class4$ utilities from the rest. The best treatment generated by the algorithm was very small only recommended changing one of the inspection processes; i.e.

$$hidesign\_12 = F$$

The $after$ bar chart of Figure 7 shows the effect of imposing this treatment of *hidesign_12=F*[g] onto the simulator. (hidesign_12 is the code we used for full Fagan inspections for high-level designs) The treated model is much improved: it generates zero $class1$ and $class2$ outputs and, in 80% of cases, generates $class4$ outputs. Further, the improvement was achieved without having to control the inspection policies in the other phases.

If we were not concerned with finding minimal solutions, TAR2 would have been called again on data generated from the inspection model, after the inputs have been constrained to *hidesign_12=F*. This iterative process could repeat until it was shown that further constraints did not improve the output. Such *interactive treatment learning* has been applied on other models (e.g. see [19]). However, for the sake of exposition, the execution of this model is not explored further.

In terms of the discussion in §3, an important feature of *hidesign_12=F* is that it is stable despite uncertainty in other parameters. Despite large scale variation of all other parameters in this model, this treatment yielded the effect seen in Figure 7. In terms of supporting commercial practices, this is a very useful result. Large corporations may have little ability to influence the practices and processes

[g] In this data set, each attribute is labeled with its column number so *hidesign_12* appears in the twelfth column the input.

C4.5 and CART are *iterative dichotomization* learners that seek the best attribute value *splitter* that most simplifies the data that fall into the different splits. Each such splitter becomes a root of a tree. Sub-trees are generated by calling iterative dichotomization recursively on each of the splits.

CART is defined for continuous target concepts and its *splitters* strive to reduce the standard deviation of the data that falls into each split. C4.5 is defined for discrete class classification and uses an information-theoretic measure to describe the diversity of classes within a data set.

A leaf generated by CART stores the average value of the class selected by the branch while a leaf generated by C4.tree stores the most frequency class. Hence, C4.5 is called a *decision tree* learner while CART is called a *regression tree*.

Fig. 8. About C4.5 and CART

of their satellite organizations or contractors. Hence, they often have to carefully select what policies to implement across the company. In terms of Equation 5, the treatment learned in this example representing the *least action* that offers the *most reward*.

Also, in terms of advocating treatment learning, the most important feature of this example is what is *missing*. To learn its treatments in this case study, TAR2 imported samples with 51 variables (50 inputs and one utility score). It then generated treatments, the best of which used only one of the inputs.

There are other commonly used data miners, such as C4.5, CART, and linear regression to which we compare TAR2. The operation and application of these are briefly summarized here:

- C4.5 and CART use the *iterative dichotimzation* algorithm described in Figure 8.
- Linear regression tries to fit one straight line through the observed values. The line offers a set of predicted values and the distance from these predicted values to the actual values is a measure of the error associated with that line. Linear regression tools such as the least squares regression package search for a line that minimizes that sum of the squares of the error.
- C4.5 predicts for discrete class symbols (e.g. *class1, class2, class3, class4*) so this algorithm used the same data as TAR2.
- Linear regression and CART make numeric predictions. These algorithms used the TAR2 data with the *classN* symbols replaced by raw numerics of Equation 5.

These data miners can be used to analyze similar models. However, they generally are far less succinct. For example, when the inspection data of this case study was passed to C4.5 [56] the decsion tree that was learned has fifty (50) nodes on seven (7) levels. A regression tree learned from CART has 24 nodes and four (4) levels. A typical node consists of five (5) to nineteen(19) terms of the form “FFFN” denoting the use of no inspections for code but full Fagan inspections for all earlier phases. A linear regression tree learned from this model’s data produced a similarly complex model.

A comparison of the output from TAR2 and C4.5 and results from CART and linear regression demonstrate that standard methods of summarizing data (linear regression, decision trees, regression trees) can generate much larger theories than treatment learning. The reason for this is very simple. Theories learned from iterative dichotomization describe the features that separate all of the target variables. However, treatments from TAR2 just describe the minimal deltas *between* preferred and undesirable targets.

Another advantage of treatment learning is that it is much easier to derive actions from treatments than from the standard methods described here. To be sure, decision trees can be analyzed to find those branch values that most often select preferred classes while most often discarding undesired classes. (The initial TAR2 prototype was such a post-processor). However, TAR2 achieves the same result directly without the need to interface to another learner.

6.2. *Studying the Capability Maturity Model (CMM)*

The previous study explored a numeric model where all the influences were precisely specified. This second case study takes a numeric model and adds a large degree of uncertainty in the numerics. This second study shows that, even in presence of large degrees of uncertainty, TAR2 can still find useful treatments.

An important feature of this second study is that it analyzes a class of models that can defeat standard methods. The model contains dozens of if-then rules; i.e. it is neither linear nor continuous: small changes in the environment can lead to "cliffs" where the model behavior changes abruptly. Also, the model contains non-deterministic choices (see the *rany* operator, discussed below) and so its behavior can be highly noisy.

This study uses a rule-based model of the costs and benefits model of the Capability Maturity Model (CMM) level 2 (hereafter, CMM2) [54, p. 125-191]. We elected to study CMM2 since, in our experience, many organizations can achieve at least this level of software process capability. CMM2 is less concerned with issues of, for example, which design pattern to apply, than with what overall project process to use. Improving CMM2-style decisions is important since, in early software life cycle, many CMM2-style decisions affect the resource allocation for the rest of the project.

In this model, CMM2 was encoded using the JANE propositional rule-based language [42]. JANE's rules take the form $Goal\ if\ SubGoals$ such as the one shown in Figure 9.

JANE is a backward chaining language: to prove a *Goal*, JANE tries to find rules that prove each of the *SubGoals*. Each *SubGoal* contributes some *Cost* and *Chances* to the *Goal*. JANE's *Chances* define the extent to which a belief in one vertex can propagate to another. *Cost*s let an analyst model the common situation where some of the *Cost* of some procedure is amortized by reusing its results many

```
stableRequirements
    if   effectiveReviews
    and requirementsUsed
    and sEteamParticipatesInPlanning
    and documentedRequirements
    and sQAactivities
    and (reviewRequirementChanges
          rany softwareConfigurationManagement
          rany baselineChangesControlled
          rany workProductsIdentified
          rany softwareTracking
        ).
```

Fig. 9. Part of CMM2, encoded in the JANE language.

times. Hence, the *first* time we use a proposition, we incur its *Cost* but afterwards, that proposition is free of charge.

The *Cost* and *Chances* of a proposition are either provided by the JANE programmer or computed at runtime via a traversal of the rules:

- When searching *X if not A*, the *Chances* of *X* are *1-Chances(A)* and $Cost(X) = Cost(A)$.
- When searching *X if A and B and C*, the *Chances* and *Costs* of *X* are (respectively) the product of the chances and the sum of the costs of *A,B,C*.
- When searching *X if A or B or C*, then the *Cost* and *Chances* of *X* are taken from the first member of *A,B,C* that is satisfied.

These *and*, *or*, *not* operators can be insufficient to capture the decision making of business users. For examples, in our experience, business users often select CMM2 options in a somewhat arbitrary manner. To model this, JANE includes a *rany* operator (short for "random any"):

- The *rany* operator is like *or* except that (e.g.) *X if A rany B rany C* succeeds if some random number of *A,B,C* (greater than one) succeeds. Unlike *and*, *or* which explore their operands in a left-to-right order, *rany* explores its *SubGoals* is a random order. If at least one succeeds, then the *Cost* and *Chances* of *X* is the sum and product (respectively) of the *Cost* and *Chances* of the satisfied members of *A,B,C*.

Rany is useful when searching for subsets that contribute to some conclusion. For example, the JANE rule in Figure 9 offers several essential features of *stableRequirements* plus several optional factors relating to monitoring change in evolving projects – the essential features are *and*-ed together while the optional factors are *rany*-ed together.

Figure 9 includes 11 propositions. Our model of CMM2, written in JANE, has 55 propositions ($range = \{t, f\}$). Of those 55 propositions, 27 were identified by our users as actions that could be changed by managers (see Figure 10).

Apart from *rany*, JANE supports one other mechanisms for exploring the space of possibilities within CMM2. When defining *Cost*s and *Chances*, the programmer can supply a *range* and a *skew*. For example:

```
goodUnitTesting and cost = 1 to +5
```

defines the *cost* of *goodUnitTesting* as being somewhere in the range 1 to 5, with the mean skewed slightly towards 5 (denoted by the "+").

baselineAudits, baselineChangesControlled, changeRequestsHandled, changesCommunicated, configurationItemStatusRecorded, deviationsDocumented, documentedDevelopmentPlan, documentedProjectPlan, earlyPlanning, formalReviewsAtMilestones, goodUnitTesting, identifiedWorkProducts, periodicSoftwareReviews

planRevised, requirementsReview, requirementsUsed, reviewRequirementChanges, risksTracked, SCMplan, SCMplanUsed, SElifeCycleDefined, SEteamParticipatesInPlanning, SEteamParticipatesOnProposal, SQAauditsProducts, SQAplan, SQAplanUsed, SQAreviewActivities, workProductsIdentified

Fig. 10. Management actions in the CMM2 model. SQA= software quality assurance and SCM= software configuration management)

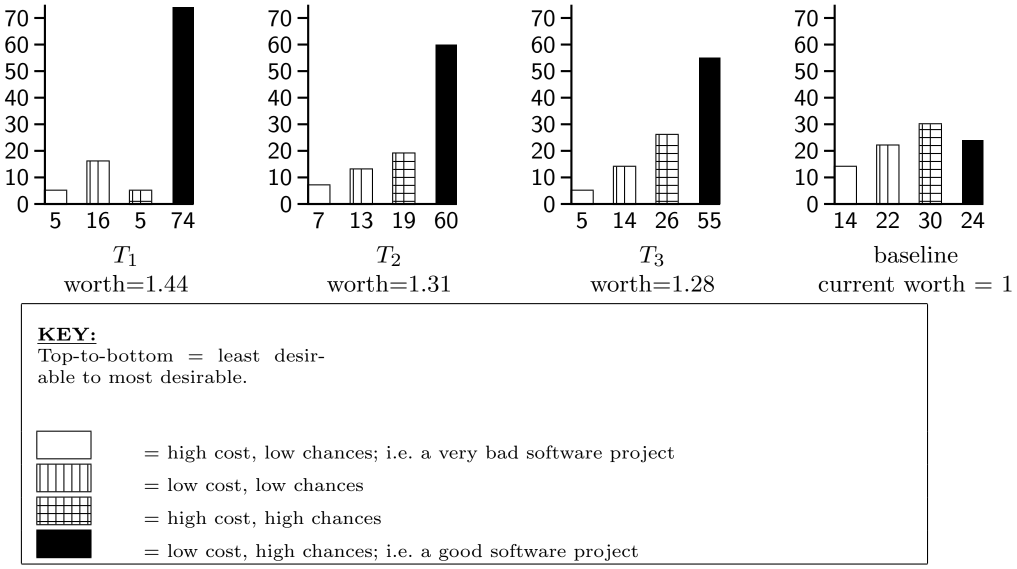


Fig. 11. Ratios of different software project types seen in four situations.

Similarly, while all the $Chances$ values were based on expert judgment, their precise value is subjective. Hence, each such $Chances$ value X was altered to be a range

```
chances = 0.7*X to 1.3*X
```

During a simulation, the *first* time a $Cost$ or $Chance$ is accessed, it is assigned randomly according to the range and skew. The assignment is cached so that all subsequent accesses use the same randomly generated value. After each simulation, the cache is cleared. After thousands of simulations, JANE can sample the "what-if" behavior resulting from different assignments within the range and many different $rany$ choices.

Data from 2000 simulations was passed from the CMM2 model to TAR2. Each simulation was classified into one of four classes:

- *class=0*: High cost, low chance;
- *class=1*: Low cost, low chance;
- *class=2*: High cost, high chance;
- *class=3*: Low cost, high chance.

That is, our preferred projects are cheap and highly likely while expensive, low odds projects are to be avoided.

Figure 11 shows three sets of actions learned by TAR2. The right-hand-side histogram shows the baseline distributions seen in the 2000 simulations. The other histograms show how those ratios change after applying the treatments learned by TAR2; The *worth* of each option is a reflection of the proportion of good and bad projects, compared to the baseline, i.e. ($worth$(*baseline*) $= 1$). Note that as $worth$ increases, the proportion of preferred projects also increases.

Figure 12 shows the three best treatments (T_1, T_2, T_3) found using this technique (and Figure 11 compared the effects of these treatments to the untreated examples). Note that the values of each attribute are reported using the tags *no*, *lower*, *middle*, or *upper*. In treatment learning, continuous attribute ranges are divided into N-discrete bands based on percentile positions. For N=3, we can name the bands *lower*, *middle*, *upper* for the lower, middle, and upper 33% percentile bands.

In Figure 12, the treatments are advising to lower the cost of:

- *Using requirements:* This could be accomplished by (e.g.) sharing them around the development team in some searchable hypertext format
- *Performing formal reviews at milestones:* This could be accomplished by (e.g.) using ultra-lightweight formal methods such as proposed by Leveson [36].
- *Performing good unit testing:* This could be accomplished by (e.g.) hiring better test engineers.

An interesting feature of Figure 12 is what is *missing*:

- None of the treatments proposed adjusting the $Chances$ of any action. In this

T_1**:** *requirementsUsed.Cost=lower and*
not periodicSoftware-Reviews and
formalReviewsAtMilestones.Cost=lower

T_2**:** *requirementsUsed.Cost=lower and*
goodUnitTesting.Cost=middle and
formalReviewsAtMilestones.Cost=lower

T_3**:** *goodUnitTesting.Cost=lower and*
periodicSoftwareReviews.Cost=middle and
formalReviewsAtMilestones.Cost=lower

Fig. 12. The three best treatments found in the CMM2 model.

study, changing $Cost$ will suffice.

- Of the 27 actions listed in in Figure 10, only the four underlined actions appear in the top three treatments. That is, management commitment to undertake 27-4=23 of the actions is less useful than changing $formalReviewsAtMilestones$, $goodUnitTesting$, $periodicSoftwareReviews$, and $requirementsUsed$
- The value *not* in T_1 is a recommendation against *periodicSoftwareReviews* (plus lowering the costs of using requirements and formal reviews at milestones). Note that if *periodicSoftwareReviews* are conducted, T_3 asserts that there is no apparent need to reduce the cost of such reviews.

More generally, in a result consistent with the prior studies, despite the uncertainties introduced by *rany* and the *cost*/*chances* ranges, TAR2 found a small number of CMM2 process options that have a significant impact on the project.

Note that the conclusions of Figure 12 are not general to all software projects. The $Chances$ values used in this study came from some local domain knowledge about the likelihood that process change A will effect process change B. The $Cost$ values were domain-specific as well. In other organizations, with different work practices and staff, those $Chances$ and $Cost$ values could be very different.

7. Conclusion

Understanding model configuration options means understanding how input choices affect output scores. That understanding is complex for a certain class of *hard models*, i.e., those with high dimensionality models that are non-linear, non-continuous and built in domains with much noise or other uncertainties, and where managers have limited control over all model inputs.

Hard modeling problems may defeat standard methods. Visualization cannot handle very large dimensionality. Analytical methods such as an eigenvector study offer spurious results if the parameters of the variables are uncertain. Other standard automatic methods may be defeated by “cliffs” in non-linear models where the association between inputs and outputs changes abruptly. Although it is true that data mining methods can handle non-linear models and scale to very large dimensionality, these data mining techniques (neural nets, genetic algorithms, C4.5, CART, etc.) often yield models that are incomprehensible to humans.

TAR2 is a special kind of data miner that produces very succinct output. It assumes that within models there exists a small number of key variables that control the rest. There is much evidence for this assumption. The mathematics of clumps and collars promises that models naturally contain structures that greatly restrict the space of possible model behaviors. TAR2 is a data miner designed to exploit such collars and clumps. It is a minimal contrast set learner that returns a "treatment"; i.e. a minimal, most influential set of deltas between different classes of outcomes. The case studies in this paper demonstrate that a minimal list of the differences between concepts can be *much smaller* than a detailed description of all aspects of a concept. For models where TAR2 can generate succinct summaries, its algorithm can significantly improved searched-based methods of data mining.

TAR2 addresses the hard modeling problems (discussed in §3) as follows:

- TAR2 offers minimal constraints on the input space and tracks the effects of those constraints, while letting all the other variables vary randomly. Hence, its proposed solutions are not brittle to changes outside the treatments.
- Since it only references a subset of the model inputs, it is a dimensionality reduction tool. In this report, we offered examples where TAR2 reasoned over 100-variable inputs spaces. Elsewhere, we have run it on data sets with over 250 variables. In all cases seen to date, it reduces those spaces to a handful of variables.
- Such small solutions are easier to explain and audit than solutions using all model inputs.
- Further, when managers do not have the budget or authority to control all model input variables, TAR2 can offer them a minimal set which they can use to focus their resources.
- TAR2 has been applied to models with large amounts of noise. For example, in a TAR2-style analysis, we often explore what happens to the solutions when the variance on the model variables increases. Such studies return statements of the form "the solutions offered by this analysis hold for variances up to the following critical threshold values, after which we do not know how to control this domain".

Treatment learning is not indicated for low-dimensionality linear continuous models built in domains that have no noise or other uncertainties, and where managers have full control over all model inputs. Other reasons not to use our tools include when where there is no need to explain or audit models, where reducing model variance is not valuable, and when there exists budgets for building and using maximal models.

As to further work, there is no reason to polarize the SE modeling field into "traditional methods" versus "treatment learning". Much could be achieved by combining the two techniques. For example, many of the methods described in §3 suffered from the curse of dimensionality. TAR2 could be used as a fast dimensionality reduction tool that could focus a data visualization environment or a sensitivity analysis on the parts of the input space that are most crucial. Ideally, that focusing need not wait till the simulation terminates. In *incremental treatment learning*, TAR2 offers feedback during a simulation run into order the guide the simulator into regions of interest. Before TAR2 can be deployed in that manner, it must be optimized so that it can run fast enough to keep up with the simulator. Currently, we are exploring stochastic methods for that optimization.

References

1. A.T. Acree. *On Mutations*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.
2. J. Aguilar-Ruiz, I. Ramos, J.C. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, December 2001.
3. A. Avritzer, J.P. Ros, and E.J. Weyuker. Reliability of rule-based systems. *IEEE Software*, pages 76–82, September 1996.
4. S.B. Bay and M.J. Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, 1999. Available from *http://www.ics.uci.edu/ pazzani/Publications/stucco.pdf* .
5. *J.M. Bieman and J.L. Schultz. An empirical evaluation (and specification) of the all-du-paths testing criterion.* Software Engineering Journal*, 7(1):43–51, 1992.*
6. *C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 1998. URL: http://www.ics.uci.edu/ mlearn/MLRepository.html* .
7. *D.S. Boning and P.K. Mozumder. Doe/opt: a system for design of experiments, response surface modeling, and optimization using process and device simulation.* IEEE Transactions on Semiconductor Manufacturing*, 7(2):233–244, May 1994.*
8. *T.A. Budd.* Mutation analysis of programs test data*. PhD thesis, Yale University, 1980.*
9. *T. Burkleaux, T. Menzies, and D. Owen. Lean = (lurch+tar3) = reusable modeling tools. In* Proceedings of WITSE 2005*, 2004. Available from http://menzies.us/pdf/04lean.pdf* .
10. *C.H. Cai, A.W.C. Fu, C.H. Cheng, and W.W. Kwong. Mining association rules with weighted items. In* Proceedings of International Database Engineering and Applications Symposium (IDEAS 98)*, August 1998. Available from http://www.cse.cuhk.edu.hk/ kdd/assoc_rule/paper.pdf* .
11. *Yuanfang Cai and Kevin J. Sullivan. Simon: modeling and analysis of design space structures. In* ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 329–332, New York, NY, USA, 2005. ACM Press.*
12. *A. Childs, J. Greenwald, G. Jung, M. Hoosier, and John Hatcliff. Calm and cadena: Metamodeling for component-based product-line development.* IEEE Computer*, 39(2), Feburary 2006. Available from http://projects.cis.ksu.edu/docman/view.php/7/129/CALM-Cadena-IEEE-*

Computer-Feb-2006.pdf

.

13. *J. Clarke, J.J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem.* IEE Proceedings-Software*, 150(3):161–175, 2003.*
14. *J. J. Dolado. A validation of the component-based method for software size estimation.* IEEE Transactions of Software Engineering*, 26(10):1006–1021, 2000.*
15. *J. J. Dolado. On the problem of the software cost function.* Information and Software Technology*, 43:61–72, 2001.*
16. *M.J. Druzdzel. Some properties of joint probability distributions. In* Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 187–194, 1994. Available from http://www.pitt.edu/AFShome/d/r/druzdzel/public/html/abstracts/uai94.html*

   .
17. *M. Fagan. Advances in software inspections.* IEEE Trans. on Software Engineering*, pages 744–751, July 1986.*
18. *Martin Feather and Steve Cornfordi. Quantitative risk-based requirements reasoning.* Requirements Engineering Journal*, 8(4):248–265, 2003.*
19. *M.S. Feather and T. Menzies. Converging on the optimal attainment of requirements. In* IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002. Available from http://menzies.us/pdf/02re02.pdf*

   .
20. *B. Feldman, P. Compton, and G. Smythe. Hypothesis Testing: an Appropriate Task for Knowledge-Based Systems. In* 4th AAAI-Sponsored Knowledge Acquisition for Knowledge-based Systems Workshop Banff, Canada*, 1989.*
21. *B. Feldman, P. Compton, and G. Smythe. Towards Hypothesis Testing: Justin, Prototype System Using Justification in Context. In* Proceedings of the Joint Australian Conference on Artificial Intelligence, AI '89*, pages 319–331, 1989.*
22. *R. E. Filman.* Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.*
23. *M.S. Fisher and T. Menzies. Learning ivv strategies. In* HICSS'06*, 2006. Available from http://menzies.us/pdf/06hicss.pdf*

   .
24. *J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering.* IEEE Computer*, 39(2):51–58, February 2006.*
25. *M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In* GECO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358. Morgan Kaufmann, July 2002.*
26. *M. Harman and B.F. Jones. Search-based software engineering.* Journal of Information and Software Technology*, 43:833–839, December 2001.*
27. *M.J. Harrold, J.A. Jones, and G. Rothermel. Empirical studies of control dependence graph size for c programs.* Empirical Software Engineering*, 3:203–211, 1998.*
28. *C.L. Heitmeyer. Software cost reduction. In John J. Marciniak, editor,* Encyclopedia of Software Engineering*, January 2002. Available from http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.pdf*

   .
29. *Gerhard J. Holzmann. Algorithms for automated protocal verification.* ATT Technical Journal*, 69(2):32–44, 1990.*

30. *Y. Hu. Treatment learning: Implementation and application. Master's thesis, Department of Electrical Engineering, University of British Columbia, 2003. Masters Thesis.*
31. *Y. Ishida. Using global properties for qualitative reasoning: A qualitative system theory. In* Proceedings of IJCAI '89*, pages 1174–1179., 1989.*
32. *Daniel Jackson. Alloy: a lightweight object modelling notation.* ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.*
33. *B. Jones, D. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing.* Computer Journal*, 41(2):98–107, 1998.*
34. *B. Jones, H.-H. Sthamer, and D. Eyres. Automatic structural tsting using genetic algorithms.* Software Engineering Journal*, 11:299–306, 1996.*
35. *Ron Kohavi and George H. John. Wrappers for feature subset selection.* Artificial Intelligence*, 97(1-2):273–324, 1997.*
36. *N. Leveson, S. Cha, and T. Shimall. Safety verification of ADA programs using software fault trees.* IEEE Software*, 8(7):48–59, July 1991.*
37. *R. Levins and C.J. Puccia.* Qualitative Modeling of Complex Systems: An Introduction to Loop Analysis and Time Averaging*. Harvard University Press, Cambridge, Mass., 1985.*
38. *R. Lutz. Evolving good hierarchical decomposition of complex systems.* Journal of Systems Architecture*, 47:613–634, 2001.*
39. *T. Menzies.* 21st *century AI: proud, not smug.* IEEE Intelligent Systems*, 2003. Available from http://menzies.us/pdf/03aipride.pdf*
.
40. *T. Menzies and Y. Hu. Reusing models for requirements engineering. In* First International Workshop on Model-based Requirements Engineering*, 2001. Available from http://menzies.us/pdf/01reusere.pdf*
.
41. *T. Menzies and Y. Hu. Just enough learning (of association rules): The TAR2 treatment learner. In* Artificial Intelligence Review*, 2007. Available from http://menzies.us/pdf/07tar2.pdf*
.
42. *T. Menzies and J.D. Kiper. Better reasoning about software engineering activities. In* ASE-2001*, 2001. Available from http://menzies.us/pdf/01ase.pdf*
.
43. *T. Menzies, A. Mahidadia, and P. Compton. Using causality as a generic knowledge representation, or why and how centralised knowledge servers can use causality. In* Proceedings of the 7th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop*, 1992.*
44. *T. Menzies, D. Owen, and B. Cukic. You seem friendly, but can i trust you? In* Formal Aspects of Agent-Based Systems*, 2002. Available from http://menzies.us/pdf/02trust.pdf*
.
45. *T. Menzies and J. Richardson. Making sense of requirements, sooner.* IEEE Computer*, October 2006. Available from http://menzies.us/pdf/06qrre.pdf*
.
46. *T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In M. Madravio, editor,* Soft Computing in Software Engineering*. Springer-Verlag, 2003. Available from http://menzies.us/pdf/03maybe.pdf*
.
47. *T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In* Proceedings ASE 2000*, 2000. Available from*

http://menzies.us/pdf/00ase.pdf
.

48. *A. Miller.* Subset Selection in Regression (second edition). *Chapman & Hall, 2002.*

49. *G.A. Miller. The magical number seven, plus or minus two: Some limits on our capacity for processing information.* The Psychological Review*, 63:81–97, 1956. Available from http://www.well.com/ smalin/miller.html*
.

50. *G.J. Myers. A controlled experiment in program testing and code walkthroughs/inspections.* Communications of the ACM*, 21:760–768, 9, September 1977.*

51. *D. Owen, B. Cukic, and T. Menzies. An alternative to model checking: Verification by random search of and-or graphs representing finite-state models. In* 7th IEEE International Symposium on High Assurance Systems Engineering*, volume 1, page 119, 2002.*

52. *D. Owen, T. Menzies, and B. Cukic. What makes finite-state models more (or less) testable? In* IEEE Conference on Automated Software Engineering (ASE '02)*, 2002. Available from http://menzies.us/pdf/02moretest.pdf*
.

53. *R.P. Pargas, M.J Harrold, and R. R. Peck. Test-data generation using genetic algorithms.* Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.*

54. *M.C. Paulk, C.V. Weber, B. Curtis, and M.B. Chriss.* The Capability Maturity Model: Guidelines for Improving the Software Process. *Addison-Wesley, 1995.*

55. *R. Pelanek. Typical structural properties of state spaces. In* Proceedings SPIN'04 Workshop*, 2004. Available from http://www.fi.muni.cz/ xpelanek/publications/state_spaces.ps*
.

56. *R. Quinlan. Induction of decision trees.* Machine Learning*, 1:81–106, 1986.*

57. *R. Quinlan.* C4.5: Programs for Machine Learning. *Morgan Kaufman, 1992. ISBN: 1558602380.*

58. *D.M. Raffo. Modeling software processes quantitatively and assessing the impact of potential process changes of process performance, May 1996. Ph.D. thesis, Manufacturing and Operations Systems.*

59. *L. Rela. Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, 2004.*

60. *P.S. Rosenbloom, J.E. Laird, and A. Newell.* The SOAR Papers. *The MIT Press, 1993.*

61. *A. Saltelli, K. Chan, and E.M. Scott.* Sensitivity Analysis. *Wiley, 2000.*

62. *A. L. Samuel. Some studies in machine learning using the game of checkers.* IBM Journal*, 3(3):211–229, July 1959.*

63. *Ed Seidewitz. What models mean.* IEEE Software*, 20(5):26–32, Sept.-Oct. 2003.*

64. *H. A. Simon.* Models of bounded rationality*, volume 2. MIT Press, 1982.*

65. *G.A. Smythe. Brain-hypothalmus, Pituitary and the Endocrine Pancreas.* The Endocrine Pancreas*, 1989.*

66. *N. Tracey, J. Clarke, and K. Mander. Automated program flaw finding using simulated annealing. In* International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, March 1998.*

67. *L.A. Treinish. A function-based data model for visualization, 1998. IBM Research Center, Yorktown Heights, NY. Available from http://www.research.ibm.com/people/l/lloydt/dm/function/dm_fn.htm*
.

68. *Geoffrey I. Webb, Shane Butler, and Douglas Newlands. On detecting differences be-*

tween groups. In KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 256–265, New York, NY, USA, 2003. ACM Press.*

69. J. Whittle and P. Jayaraman. Generating hierarchical state machines from use case charts. In IEEE International Conference on Requirements Engineering (RE2006)*, 2006.*

70. W.E. Wong and A.P. Mathur. Reducing the cost of mutation testing: An empirical study. The Journal of Systems and Software*, 31(3):185–196, December 1995.*