A Data Miner for Searching Model-Based Software

Tim Menzies, *Member, IEEE,* , James Kiper, *Member, IEEE,* Jeremy Greenwald, Ying Hu,
David Raffo, *Member, IEEE*, and Siri-on Setamanit

***Abstract*— Model-based software engineering allow developers to explore trade spaces more effectively. Building a useful model can be intellectually difficult. Hence, we propose a combination of human and artificial intelligence where humans proposes a set of possible influences, then a special kind of machine learner (a *minimal contrast set learner* selects the subset of the model. We show theoretically and empirically that our TAR2 minimal contrast set learner can generate very succinct conclusions from seemingly complex spaces.**

***Index Terms*— model-driven software engineering, contrast set learning, treatment learning**

I. Introduction

SOFTWARE engineering is modeling [1]. Software engineers build models at every phase of the life cycle. Some are paper-based and some are executable but all these artifacts are *models* which Mellor [2] defines to be "elements describing something (for example, a system, bank, phone, or train) built for some purpose that is amenable to a particular form of analysis, such as communication of ideas between people and machines; completeness checking; test case generation; etc."

The current ubiquity of the term *model-driven software engineering* is both a recognition that modeling has long been a central activity in software development, and an appreciation of the reality that we are not using these models as effectively as we may hope. We therefore endorse the thesis that models are currently used throughout the software development process – from requirements and design to testing and performance evaluation – and that the use of such models will only increase. We also recognize that, even with tool support, creation of practical and functional models is a resource-intensive activity, especially in terms of human intelligence. Effective use of models requires that they be allowed to evolve through the software development process. Other researchers have recognized that "designers must be able to examine various design alternatives quickly and easily among myriad and diverse *configuration possibilities*" [3].

The number of configuration possibilities within a model can be dauntingly large. A model with 20 binary choices has $2^{20} > 1,000,000$ possible configurations, far beyond the capability of human comprehension. Since 2000 we have explored sampling those configurations at random, running the resulting model, scoring the output with some oracle, then using data mining to find the configuration options that most improve model output [4], [4]–[18]. This paper synthesizes that prior work and presents new case studies. We have found that standard data miners may be inadequate for exploring model configuration possibilities. Standard data miners often yield results that still are incomprehensible to humans:

- Neural nets never generate a succinct generalization of their knowledge [19];
- The random search of genetic algorithms can produce models that are too complex to understand [20];
- Most decision/regression tree learners such as C4.5 [21] or CART [22] execute in local top-down search, with no memory between different branches. The same concept can hence be needlessly repeated many times within the tree. Consequently, such trees can be cumbersome, needlessly large, and difficult to understand.

Our premise is that the detailed, complex and arcane output generated by standard data miners is often superfluous. Firstly, many human experts can not (or will not) read complex theories learned by a data miner. Secondly, numerous empirical and theoretical results argue that, in the usual case, models are controlled by a handful of of key variables. If data miners are restricted to just returning models containing those keys, then the learned model will be very small indeed.

Hence, we propose a different kind of data miner to assist analysts in exploring the configuration possibilities within their models. Traditional machine learners like C4.5 generate classifiers that assign a class symbol to an example. Our preferred method, called *treatment learning* just generates the *differences* in the key variables between different outcomes.

The rest of this paper is structured as follows. The twin topics of model-based SE and search-based SE are introduced. Evidence is then presented that, often, the key variables that control a model are few in number. This will motivate the design of the TAR2 *treatment learner*. Case studies will then be presented showing that, in many domains, TAR2 finds a small number of controlling variables. The conclusion will be that understanding the configuration possibilities within model-based SE can be dramatically simplified via treatment learning.

Dr. Menzies is with the Lane Department of Computer Science, West Virginia University and can be reached at tim@menzies.us

Dr. Kiper is with the Computer Science and Systems Analysis Department, School of Engineering & Applied Science, Miami University, and can be reached at kiperjd@muohio.edu

Mr. Greenwald is with Computer Science, Portland State University and can be reached at jegreen@cecs.pdx.edu

Ms. Hu is a software designer in Vancouver, British Columbia and can be reached at huying_ca.@yahoo.com

Dr. Raffo is a Associate Professor in the School of Business Administration, Portland State University, and can be reached at raffod@sba.pdx.edu

Ms. Setamanit is a graduate student at Portland State University and can be reached at sirion@pdx.edu

The research described in this paper was carried out at Miami University, West Virginia University and Portland State University under contracts and sub-contracts with NASA's Software Assurance Research Program. Reference herein to any specific commercial product, process, or service by trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

Manuscript received January 1, 2007. Download an earlier draft from http://menzies.us/pdf/07exploit.pdf

II. Modeling and Search

Anthropologists argue that the ability to build abstract models is the what gives homo sapiens their competitive edge. In his article "What Models Mean" [23], Seidewitz describes the interactions and relationships among the concepts of a model, model *correctness*, model *interpretation*, a theory that allows us to deduce new statements for the model, a *modeling language*, and a model *interpretation*. 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 ..." These are the aspects of a model's meaning on which we focus in *model-based* software engineering.

Model-based software engineering (SE) is becoming increasingly important. Sendall and Kozacaynski argue that increasing productivity and reduced time-to-market for software products can accrue when "using concepts closer to the problem domain ..." via modeling [24]. Hailpern and Tarr observe that model-driven development "imposes structure and common vocabularies so that artifacts are useful for their main purpose in their particular stage in the life cycle." [25]

The utility of model-based SE is widely acknowledged. For example, the Object Management Group (OMG) has recently adopted a *model-driven architecture* framework with goals of "portability, interoperability and reusability through architectural separation of concerns." [26] Also, Microsoft has been developing a *Software Factory* concept that leverages models with the goal of automation of the software development process [27]. And, at Lockheed Martin, engineers have developed an integrated modeling method called *Model Centric Software Development* that uses "automated generation of partial implementation artifacts," reverse engineering to integrate legacy assets, and model verification and checking [28]

The importance of models and the model-driven approach is not limited to software design and UML. Many tools exist for modeling such as distributed agent-based simulations [29], discrete-event simulation [30]–[32], continuous simulation (also called system dynamics) [33], [34], state-based simulation (which includes petri net and data flow approaches) [35]–[37], logic-based and qualitative-based methods [38, chapter 20] [39], and rule-based simulations [40]. One can find models used in the requirements phase (see the DDP method and tool [41] which are a risk-based approach and visual tool to support requirements engineering), refactoring of designs using patterns [42], software integration [43], model-based security [44], and performance assessment [45].

Model are useful since humans can review/audit/improve an explicit representation of their systems. 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, [3] have developed the Constraint-Specification Aspect Weaver (C-Saw) that uses aspect-oriented approaches [46] to help engineers in the process of model transformation. Cai and Sullivan [47] 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 [48] and SCR [49] as well as various UML tools that allow for the execution of life cycle specifications (e.g. the CADENA scenario editor [50]).

Recently, AI has been successful 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) [51]. 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. [52] 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." [52].

Search-based SE is most often used to optimizing software testing [53]–[56] but it has has application in numerous other areas. With Feather, we have used search-based SE for requirements analysis [41]. Other researchers [57], [58] use genetic algorithms to examine ways of modularizing software [52] or developing effort estimators [59]–[61]. In all, Rela [62] lists 123 publications were search-based methods have been applied to the above applications as well as automatic synthesis of software detect predictors; assisting in component design; developing multiprocessor schedules; re-engineering old systems into 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*; i.e. an "oracle" that scores a model configuration. Current model-based SE methods rarely offer such a function (exception: formal methods that generate temporal constraints). In our experience, generating such a fitness function is usually possible, albeit after days of work with the domain experts [63].
- Determining an appropriate set of *manipulation operators* to select future searches based on the prior searches [64].

Data mining is one way to implement automatic manipulation operators. A data miner searches through the space of possible concepts for a combination of concepts that describes some target theory [65]. Given, say, the output from a Monte Carlo simulation of a model, a 21st century data miner can sift through gigabytes of data looking for the core concepts that most select for preferred output.

III. 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. If so, then the output of data miner could be simplified to just constrain the collar variables that switch the system between a few clumps. Such brevity is useful since:

- Smaller models are easier to explain (or audit).
- Large models often contain irrelevant or redundant details, which search-based methods can learn to ignore.
- Miller shows that models generally contain fewer variables have less variance in their outputs [66].
- The smaller the model, the fewer are the demands on interfaces (sensors and actuators) to the external environment. Hence, systems designed around small models are easier to use (less to do) and cheaper to build.

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 effect 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 effected 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 different branches in different 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 .

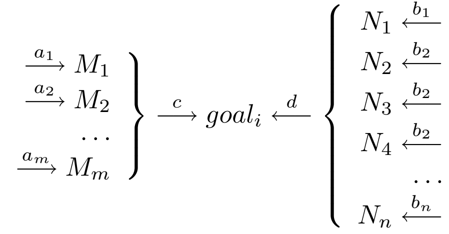
A related effect to *collars* is *clumping*. If a program has v variables with range r, then the maximum number of states is r^v. Programs $clump$ when most of those states are never $used$ at runtime; i.e. $|used|/\left(r^v\right) \approx 0$. Clumps can cause collars:

- The size of $used$ is the cardinality cross product of the ranges seen in the controllers.
- If that cardinality is large, many states will be generated and programs won't clump.
- But if that cross product 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 what states are reached at runtime.

There is much theoretical and empirical evidence for expecting that models often contain *collars* and *clumps*.

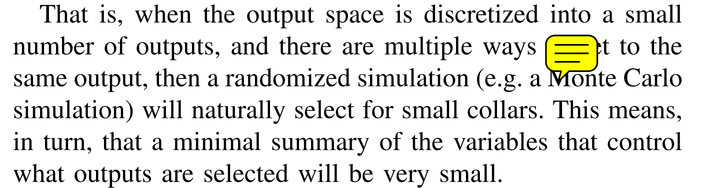
A. Theoretical Evidence

With Singh [14], we have showed 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 $b \rightarrow goal$.



Each of the terms in lower case in the above diagram represent a probability of some event; i.e. $0 \leq \{a_i, b_i, c, d, goal\} \leq 1$. 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. Under a variety of conditions, 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 [14]).



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. This means, in turn, that a minimal summary of the variables that control what outputs are selected will be very small.

As to *clumping*, Druzdel [67] 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

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

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, p_i is uniform and 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 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.

B. Empirical Evidence

Empirical evidence for clumps first appeared in the 1950s. Writing in 1959, Samuel studied machine learning for the game of checkers [68]. At the heart of his program was a 32-term polynomial ~~scoring how~~ board configurations~~;~~ e.g. *king center control* means that a king occupies one of the center positions. The program learns weights for these variable coefficients. After 42 games, the program had learned that 12 variable 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 [69].

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. This *early coverage* effect has been observed many times. In a telecommunications application, Avritzer, Ros, & 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%) [70]. 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 $\in \{10,15,\ldots 40,100\}$). 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 [71]. The same observation has been made elsewhere by Budd [72] and Acree [73].

If the space of possible execution pathways within a program are 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 [74] 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 [75] 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 [1] and small SCC quotient"[2] [76]. 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 [77].

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" [76]. 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 [78] who have difficulty designing overly-complex models.

IV. Data Mining with Collars and Clumps

The TAR2 *treatment learner* [79] is a data miner that is specialized for generating models containing only collar variables. TAR2 finds the difference between classes. Formally, the algorithm is a *contrast set learner* [80], [81] that uses *weighted classes* [82] 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 very simple to implement. Consider a log of golf playing behavior shown in Figure 1. This log contains four attributes (outlook, temperature, humidity, wind) and 3 classes (none, some, lots) that convey the amount of golf played. We recommend an exponential scoring system for the classes, starting at two[3].

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

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

[3] 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 [8].

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. 1. A log of some golf-playing behavior.

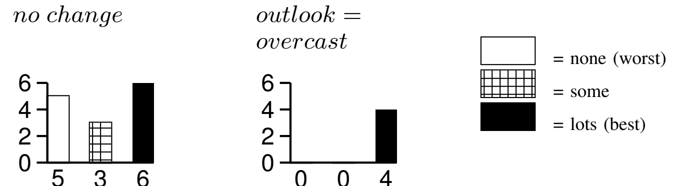


Fig. 2. Finding treatments that can improve golf playing behavior. With no treatments, we only play lots of golf in $\frac{6}{5+3+6} = 57\%$ of cases. However, assuming *outlook=overcast*, we play golf lots of times in 100% of cases.

For example, our golfer could weight the classes in Figure 1 as *none=2* (worst), *some=4*, *lots=8* (best).

TAR2 seeks attribute ranges that are frequently in the highly weighted classes and rare 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 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:

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

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

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

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. 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 (default value= 1; maximum useful value yet found= 7).

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 1 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 2 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.

In practice, despite the $\frac{n!}{s!(n-s)!}$ search, TAR2 scales well. Figure 3 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.

V. 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.

A. Inspection Policies

The first case study contrasts treatment learning with traditional learners. It will be seen that treatment are dramatically

		# 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. 3. Runtimes for TAR2 on different domains. First 6 data sets come from the UC Irvine machine learning data repository [83]; "cocomo" comes from a COCOMO software cost estimation model [4]; "pilot" comes from the NASA Jet Propulsion Laboratory [6].

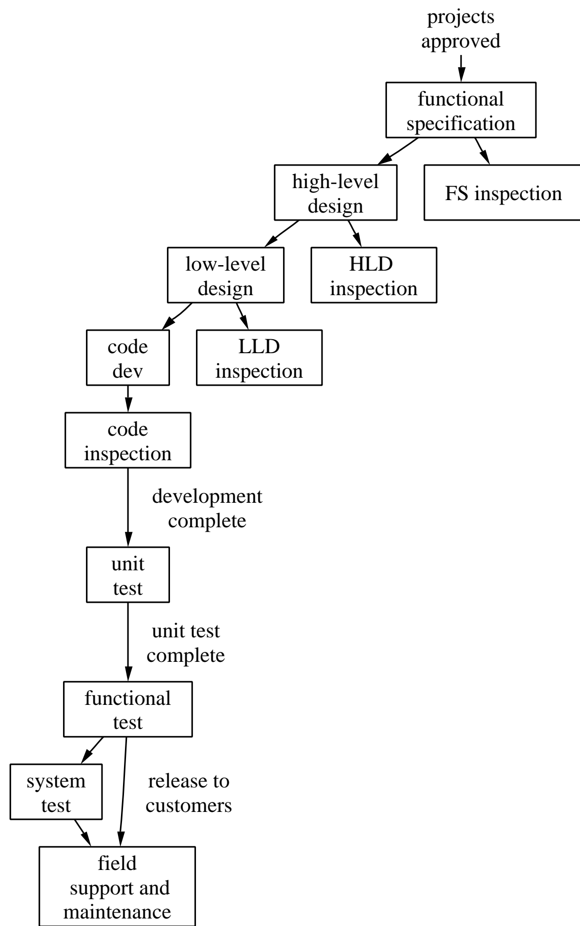


Fig. 4. High-level block diagram of a discrete event model of one company's software process.

smaller, and more understandable, than the model learned by standard data miners.

Figure 4 offers a high-level view of a quantitative software process model [84]. 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 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:

$$utility = \quad 40 * (14 - quality) \; + \; 320 * (70 - expense) + 640 * (24 - duration) \tag{2}$$

The justification for this style of utility function is discussed in detail in [84]. 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 Figure 4, 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* [85] is a seven step process with pre-determined roles for inspection participants. For the company studied by Raffo, the *defect detection capability*[4] of their full Fagan inspections was $TR(0.35, 0.50, 0.65)$[5]. 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 $\{min, mode, max\}$ = $\{0.13, , 0.21, 0.30\}$.

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 2. These utilities were discretized into four classes, of approximately equal frequency:

- *class1* : value of Equation 2 $<$ 9843
- *class2* : 9843 $\leq$ value of Equation 2 $<$ 10698
- *class3* : 10698 $\leq$ value of Equation 2 $<$ 11664
- *class4* : 11664 $\leq$ value of Equation 2 $\leq$ 14755

The $before$ bar chart in Figure 5 shows the frequency of these classes in the untreated model. Note that many (40%) of the samples generate the lowest $class1$ utility.

[4]Defect detection capability is the percentage of defects that are latent in the artifact that is being inspected that are detected.

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

Fig. 5. Utility frequency in 12800 samples, *before* and *after* treatment.

```
corErr_gg50 <= 72: class1
corErr_gg50 > 72
|   corErr_gg50 <= 154
|   |   spec_gg2 = b
|   |   |   hiDist_gg13 = 0
|   |   |   |   corErr_gg11 <= 17: class1
|   |   |   |   corErr_gg11 > 17: class2
|   |   |   hiDist_gg13 = tr
|   |   |   |   corErr_gg31 <= 18: class1
|   |   |   |   corErr_gg31 > 18
|   |   |   |   |   inspEff_gg19 <= 33.742185: class1
|   |   |   |   |   inspEff_gg19 > 33.742185: class2
|   |   spec_gg2 = f
|   |   |   corErr_gg21 <= 22
|   |   |   |   detCap_gg38 <= 0.563188
|   |   |   |   |   inspDur_gg10 <= 117.043523
|   |   |   |   |   |   corErr_gg50 <= 120: class1
|   |   |   |   |   |   corErr_gg50 > 120
|   |   |   |   |   |   |   inspDur_gg40 <= 292.445892: class2
|   |   |   |   |   |   |   inspDur_gg40 > 292.445892: class1
|   |   |   |   |   inspDur_gg10 > 117.043523: class1
|   |   |   |   detCap_gg38 > 0.563188: class2
|   |   |   corErr_gg21 > 22: class2
|   |   spec_gg2 = n
|   |   |   inspEff_gg29 <= 336.924853: class2
|   |   |   inspEff_gg29 > 336.924853: class1
|   |   spec_gg2 = w
|   |   |   corErr_gg11 <= 3: class1
|   |   |   corErr_gg11 > 3
|   |   |   |   corErr_gg11 <= 4
|   |   |   |   |   inspEff_gg9 <= 32.423212: class2
|   |   |   |   |   inspEff_gg9 > 32.423212
|   |   |   |   |   |   inspDur_gg10 <= 35.939846
|   |   |   |   |   |   |   detCap_gg8 <= 0.110608: class2
|   |   |   |   |   |   |   detCap_gg8 > 0.110608: class1
|   |   |   |   |   |   inspDur_gg10 > 35.939846: class2
|   |   |   |   corErr_gg11 > 4
|   |   |   |   |   detCap_gg8 <= 0.145107
|   |   |   |   |   |   inspEff_gg9 <= 49.227938: class1
|   |   |   |   |   |   inspEff_gg9 > 49.227938
|   |   |   |   |   |   |   detCap_gg8 <= 0.131955: class2
|   |   |   |   |   |   |   detCap_gg8 > 0.131955: class1
|   |   |   |   |   detCap_gg8 > 0.145107
|   |   |   |   |   |   inspEff_gg9 <= 33.741217: class1
|   |   |   |   |   |   inspEff_gg9 > 33.741217: class2
|   corErr_gg50 > 154
|   |   detCap_gg8 <= 0.618237: class2
|   |   detCap_gg8 > 0.618237
|   |   |   inspDur_gg30 <= 137.500433: class3
|   |   |   inspDur_gg30 > 137.500433: class2
```

Fig. 6. A decision tree learned via C4.5.

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 5 shows the effect of imposing

C4.5 and CART are well-understood and widely-cited *iterative dichotomization* learners. Such learners:

- 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.

Another difference between C4.5 and CART is how the leaf nodes summarize the examples that fall down the branch to that leaf. 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. 7. About C4.5 and CART

this treatment of *hidesign_12=F*[6] (full Fagan inspections for high-level designs) onto the simulator. 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.

In terms of advocating treatment learning, the most important feature of this example is what is *missing*. To learn its treatmetns, 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. Other data miners, working on the same data, are far less succinct. For example, Figure 6 show the results of passing the inspection model data to C4.5 [21] and an appendix to this document shows the output from the CART and linear regression learners:

- C4.5 and CART use the *iterative dichotimzation* algorithm described in Figure 7.
- 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 2.

A comparison of the output from TAR2, C4.5 (in Figure 6), and results from CART and linear regression (see appendix) 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.

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

However, treatments 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 shown in Figure 6 and the appendix. To be sure, decision trees can be analyzed to find branch values that most selected for preferred classes while most 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.

B. Studying the CMM

The previous studied 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.

This study uses a rule-based model of the costs and benefits model of CMM level 2 (hereafter, CMM2) [86, p125-191]. We elected to study CMM2 since, in our experience, many organizations can achieve at least this level. CMM2 is less concerned with issues of (e.g.) which design pattern to apply, than with what overall project structure should be implemented. 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.

CMM2 was encoded using the JANE propositional rule-based language [9]. JANE's rules take the form $Goal\ if\ SubGoals$ such as the one shown in Figure 8.

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

Fig. 8. Part of CMM2, encoded in the JANE language.

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. $Costs$ let an analyst model the common situation where some of the $Cost$ of some procedure is amortized by reusing its results many 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 select CMM2 options, often 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 8 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 8 includes 11 propositions. Our model of CMM2, written in JANE, has 55 proposition ($range = \{t, f\}$). Of those 55 propositions, 27 were identified by our users as actions that could be changed by managers (see Figure 9).

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 "+").

Similarly, while all the $Chances$ values were based on expert judgment, their precise value is subjective. Hence, each

baselineAudits, baselineChangesControlled, changeRequestsHandled, changesCommunicated, configurationItemStatus-Recorded, deviationsDocumented, documentedDevelopmentPlan, documentedProjectPlan, earlyPlanning, formalReviewsAtMilestones, goodUnitTesting, identifiedWorkProducts, periodicSoftwareReviews

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

Fig. 9. Management actions in the CMM2 model. SQA= software quality assurance and SCM= software configuration management)

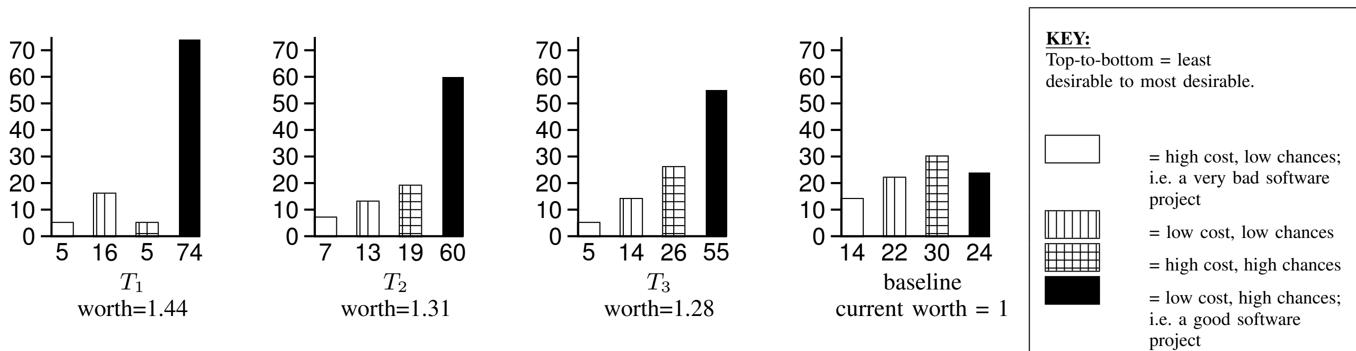


Fig. 10. Ratios of different software project types seen in four situations.

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=4*: Low cost, high chance.

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

Figure 10 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 11 shows the three best treatments (T_1, T_2, T_3) found using this technique (and Figure 10 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 11, 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 search-able hypertext format
- *Performing formal reviews at milestones:* This could be accomplished by (e.g.) using ultra-lightweight formal methods such as proposed by Leveson [87].

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. 11. The three best treatments found in the CMM2 model.

- *Performing good unit testing:* This could be accomplished by (e.g.) hiring better test engineers.

An interesting feature of Figure 11 is what is *missing*:

- None of the treatments proposed adjusting the $Chances$ of any action. In this study, changing $Cost$ will suffice.
- Of the 27 actions listed in in Figure 9, 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 on $formalReviewsAtMilestones$, $goodUnitTesting$, $periodicSoftwareReviews$, 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 is saying 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 11 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.

C. Other Case Studies

Treatment learning has been applied to spacecraft design to find how to cover more requirements, reduce risk, at the least

cost [6], [12]. It has also been applied to software process control using:

- A Chung-Mypolopous soft-goal graph to find better coverage of the non-functional requirements [13].
- COCOMO effort and risk models models to find options selecting for lower effort and fewer risks [4];
- COCOMO effort, risk, and defect prediction models models to find project options selecting for lower effort and fewer threats and lower defects [15];
- Qualitative inference diagrams to find requirements selecting for higher quality [16].
- The NASA SILAP model (that selects V&V tasks in order to most lower risks) [17];

Treatment learning has also been applied to:

- Finite state machines to find topologies that reduce the CPU cost of applying formal methods [10], [18].
- Models of the global economy so study methods of extending human life expectancy [7];
- Maximizing whiskey production [5];

In all the case studies explored by TAR2, the same three observations were made:

- Treatment learning can find very small treatments, even for seemingly complex models;
- These treatments can be far smaller that models generated by standard data miners.
- Despite uncertainties or variabilities in the model, TAR2 was able to find effective treatments that selected for preferred model output (but the less uncertainty or model variability, the smaller the variance in TAR2's predicted output for the treated model).

VI. Related Work

Elsewhere [14] we have discussed the connection of treatment learning to soft computing methods such as fuzzy logic, genetic algorithms, and neural nets. Approximate, heuristically generated models would be adequate to control models if those models contain clumps and collars. That is, the widespread presence of clumps and collars would explain the success of soft computing.

Treatment learners share the same goals as, but uses different methods from, *sensitivity analysis* [88] and *design of experiments* (DOE) methods [89] that seek out the key factors that most influence a model (also, recommended settings for those key factors are generated). A canonical sensitivity analysis method might be to compute eigenvector of a linear system in order to understand its long-term temporal behaviour. However, for models where the collar and clump assumption holds, a detailed exploration of the nuances of the model's states may yield little more information than TAR2's treatments.

Treatment learning would be preferred to numerical sensitivity analysis when the goal is a succinct policy statement that can be quickly explained to business users. Also, treatment learning might be a useful pre-processor to sensitiviy analysis in order to focus the sensitivity analysis on the parts of the model that are most crucial. On the other hand, TAR2 is not the tool of choice when seeking intricate optimizations of complex non-linear systems with massive feedback. For such optimizations, standard statistical sensitivity methods are generally preferred to treatment learning.

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 that the model is sensitive to. So, in this regard, DOE and TAR2 are analogous. However, TAR2 augments the usual DOE rig with the generation of a new extremely parsimonious approximate from an initial, far more complex, model.

It is insightful to contrast treatment learning with automatic formal methods. Treatment learner offers coarse-grained heuristic controllers for models with a wide range of internal parameter settings. Automatic formal methods, on the other hand, rigorously explore the structure of small models that are precisely specified (albeit with some non-determinism). Hence they are tools to be used for *different tasks* in the life cycle of a model.

Early in the software life cycle, engineers are concerned with exploring a large space of possible models or possible model configurations. Model-based tools needed for such *option assessment* (e.g. TAR2) may be very different from model-based tools for *fine-tuning* and verifying a final model such as SPIN [90]. Certainly, for mission and safety critical models, it is essential that such rare but dangerous bugs be located and fixed. However, while much of the model remains under debate, the rigorous search of, say, SPIN can be rendered obsolete by the next revision to the model.

VII. Conclusion

Controlling some models may be much simpler than previously thought. Models are written by people; people have cognitive limits; therefore models written by people aren't always complex. Also, regardless of who writes a model, the mathmematics of clumps and collars promises that models naturally contain structures that greatly restrict the space of possible model behaviors.

It is possible to design data miners to exploit such collars and clumps. The case studies in this paper show 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. Human designers could avoid wasting time on superfluous details in that models could be iteratively built by humans, then pruned by minimal contrast set learning. In this way, human-based design intelligence could be augmented in a cost-effective manner via artificial intelligence.

Treatment learning will fail when models don't contain clumps and collars. Such models can't be controlled by a small number of key variables– in which case TAR2's tiny rules will never be useful. However, this paper has presented theoretical and empirical evidence to suggest that collars and clumps can be expected in many models. Hence, treatment learning should be applicable to much of model-based SE, particular when large numbers of configuration possibilities are being discussed.

REFERENCES

[1] D. C. Schmidt, "Model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 25–31, February 2006.

[2] S. Mellor, A. Clark, and T. Futagamii, "Model-driven development - guest editor's introduction," *IEEE Software*, vol. 20, no. 5, pp. 14– 18, Sept.-Oct. 2003.

[3] J. Gray, Y. Lin, and J. Zhang, "Automating change evolution in model-driven engineering," *IEEE Computer*, vol. 39, no. 2, pp. 51–58, February 2006.

[4] 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.

[5] 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.

[6] M. 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.

[7] D. Geletko and T. Menzies, "Model-based software testing via treatment learning," in *IEEE NASE SEW 2003*, 2003, available from http://menzies.us/pdf/03radar.pdf.

[8] Y. Hu, "Treatment learning: Implementation and application," Master's thesis, Department of Electrical Engineering, University of British Columbia, 2003, masters Thesis.

[9] T. Menzies and J. Kiper, "Better reasoning about software engineering activities," in *ASE-2001*, 2001, available from http://menzies.us/pdf/01ase.pdf.

[10] 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.

[11] D. R. T. Menzies, J. Smith, "When is pair programming better?" 2003, available from http://menzies.us/pdf/04pairprog.pdf.

[12] S. L. Cornford, J. D. M. S. Feather, J. Salcedo, and T. Menzies, "Optimizing spacecraft design optimization engine development: Progress and plans," in *Proceedings of the IEEE Aerospace Conference, Big Sky, Montana*, 2003, available from http://menzies.us/pdf/03aero.pdf.

[13] E. Chiang and T. Menzies, "Simulations for very early lifecycle quality evaluations," *Software Process: Improvement and Practice*, vol. 7, no. 3-4, pp. 141–159, 2003, available from http://menzies.us/pdf/03spip.pdf.

[14] T. Menzies and H. Singh, "Many maybes mean (mostly) the same thing," in *Soft Computing in Software Engineering*, M. Madravio, Ed. Springer-Verlag, 2003, available from http://menzies.us/pdf/03maybe.pdf.

[15] T. Menzies and J. Richardson, "Xomo: Understanding development options for autonomy," in *COCOMO forum, 2005*, 2005, available from http://menzies.us/pdf/05xomo_cocomo_forum.pdf. For more details, see also the longer technical report http://menzies.us/pdf/05xomo101.pdf.

[16] ——, "Making sense of requirements, sooner," *IEEE Computer*, October 2006, available from http://menzies.us/pdf/06qrre.pdf.

[17] M. Fisher and T. Menzies, "Learning iv&v strategies," in *HICSS'06*, 2006, available from http://menzies.us/pdf/06hicss.pdf.

[18] 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.

[19] G. Hinton, "How neural networks learn from experience," *Scientific American*, pp. 144–151, September 1992.

[20] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[21] R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, pp. 81–106, 1986.

[22] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, "Classification and regression trees," Wadsworth International, Monterey, CA, Tech. Rep., 1984.

[23] E. Seidewitz, "What models mean," *IEEE Software*, vol. 20, no. 5, pp. 26–32, Sept.-Oct. 2003.

[24] S. Sendall and W. Kozacaynski, "Model transformation: The heart and soul of model-driven software development," *IEEE Software*, vol. 20, no. 5, pp. 42–45, Sept.-Oct. 2003.

[25] B. Hailpern and P. Tarr, "Model-driven develpment: the good, the bad, and the ugly," *IBM Systems Journal*, vol. 45, no. 3, pp. 451–461, 2006.

[26] *MDA Guide Version 1.0.1*, Object Management Group, June 2003. [Online]. Available: http://www.omg.org/mda/presentations.htm

[27] J. Greenfield and K. Short, *Software factories : assembling applications with patterns, models, frameworks, and tools*. Wiley Publishing, Indianapolis, IN, 2004.

[28] D. Waddington and P. Lardieri, "Model-centric software development," *IEEE Computer*, vol. 39, no. 2, pp. 28–29, February. 2006.

[29] W. Clancey, P. Sachs, M. Sierhuis, and R. van Hoof, "Brahms: Simulating practice for work systems design," in *Proceedings PKAW '96: Pacific Knowledge Acquisition Workshop*, P. Compton, R. Mizoguchi, H. Motoda, and T. Menzies, Eds. Department of Artificial Intelligence, 1996.

[30] A. Law and B. Kelton, *Simulation Modeling and Analysis*. McGraw Hill, 2000.

[31] H. Harrell, L. Ghosh, and S. Bowden, *Simulation Using ProModel*. McGraw-Hill, 2000.

[32] D. Kelton, R. Sadowski, and D. Sadowski, *Simulation with Arena, second edition*. McGraw-Hill, 2002.

[33] T. Abdel-Hamid and S. Madnick, *Software Project Dynamics: An Integrated Approach*. Prentice-Hall Software Series, 1991.

[34] H. Sterman, *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill, 2000.

[35] M. Akhavi and W. Wilson, "Dynamic simulation of software process models," in *Proceedings of the 5th Software Engineering Process Group National Meeting (Held at Costa Mesa, California, April 26 - 29)*. Software engineering Institute, Carnegie Mellon University, 1993.

[36] D. Harel, "Statemate: A working environment for the development of complex reactive systems," *IEEE Transactions on Software Engineering*, vol. 16, no. 4, pp. 403–414, April 1990.

[37] R. Martin and D. M. Raffo, "A model of the software development process using both continuous and discrete models," *International Journal of Software Process Improvement and Practice*, June/July 2000.

[38] I. Bratko, *Prolog Programming for Artificial Intelligence. (third edition)*. Addison-Wesley, 2001.

[39] Y. Iwasaki, "Qualitative physics," in *The Handbook of Artificial Intelligence*, P. C. A. Barr and E. Feigenbaum, Eds. Addison Wesley, 1989, vol. 4, pp. 323–413.

[40] P. Mi and W. Scacchi, "A knowledge-based environment for modeling and simulation software engineering processes," *IEEE Transactions on Knowledge and Data Engineering*, pp. 283–294, September 1990.

[41] M. Feather and S. Cornfordi, "Quantitative risk-based requirements reasoning," *Requirements Engineering Journal*, vol. 8, no. 4, pp. 248–265, 2003.

[42] R. France, S. Ghosh, E. Song, and D. Kim, "A metamodeling approach to pattern-based moel refractoringt," *IEEE Software*, vol. 20, no. 5, pp. 52–58, Sept.-Oct. 2003.

[43] P. Denno, M. P. Steves, D. Libes, and E. J. Barkmeyer, "Model-drven integration using existing models," *IEEE Software*, vol. 20, no. 5, pp. 59–63, Sept.-Oct. 2003.

[44] J. Jrjens and J. Fox, "Tools for model-based security engineering," in *ICSE '06: Proceeding of the 28th international conference on Software engineering*. New York, NY, USA: ACM Press, 2006, pp. 819–822.

[45] S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni, "Model-based performance prediction in software development: A survey," *IEEE Transactions on Software Engineering*, vol. 30, no. 5, May 2004.

[46] R. E. Filman, *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.

[47] Y. Cai and K. 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*. New York, NY, USA: ACM Press, 2005, pp. 329–332.

[48] D. Jackson, "Alloy: a lightweight object modelling notation," *ACM Trans. Softw. Eng. Methodol.*, vol. 11, no. 2, pp. 256–290, 2002.

[49] C. Heitmeyer, "Software cost reduction," in *Encyclopedia of Software Engineering*, J. J. Marciniak, Ed., January 2002, available from http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.p%df.

[50] A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff, "Calm and cadena: Metamodeling for component-based product-line development," *IEEE Computer*, vol. 39, no. 2, Feburary 2006, available from http://projects.cis.ksu.edu/docman/view.php/7/129/CALM-Cadena-IEEE-Comp%uter-Feb-2006.pdf.

[51] J. Whittle and P. Jayaraman, "Generating hierarchical state machines from use case charts," in *IEEE International Conference on Requirements Engineering (RE2006)*, 2006.

[52] J. Clarke, 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*, vol. 150, no. 3, pp. 161–175, 2003.

[53] B. Jones, H.-H. Sthamer, and D. Eyres, "Automatic structural tsting using genetic algorithms," *Software Engineering Journal*, vol. 11, pp. 299–306, 1996.

[54] B. Jones, D. Eyres, and H.-H. Sthamer, "A strategy for using genetic algorithms to automate branch and fault-based testing," *Computer Journal*, vol. 41, no. 2, pp. 98–107, 1998.

[55] R. Pargas, M. Harrold, and R. R. Peck, "Test-data generation using genetic algorithms," *Journal of Software Testing, Verification and Reliability*, vol. 9, pp. 263–282, 1999.

[56] N. Tracey, J. Clarke, and K. Mander, "Automated program flaw finding using simulated annealing," in *International Symposium on Software Testing and Analysis*. ACM/SIGSOFT, March 1998, pp. 73–81.

[57] 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*. Morgan Kaufmann, July 2002, pp. 1351–1358.

[58] R. Lutz, "Evolving good hierarchical decomposition of complex systems," *Journal of Systems Architecture*, vol. 47, pp. 613–634, 2001.

[59] J. Aguilar-Ruiz, I. Ramos, J. Riquelme, and M. Toro, "An evolutionary approach to estimating software development projects," *Information and Software Technology*, vol. 43, no. 14, pp. 875–882, December 2001.

[60] J. J. Dolado, "A validation of the component-based method for software size estimation," *IEEE Transactions of Software Engineering*, vol. 26, no. 10, pp. 1006–1021, 2000.

[61] ——, "On the problem of the software cost function," *Information and Software Technology*, vol. 43, pp. 61–72, 2001.

[62] L. Rela, "Evolutionary computing in search-based software engineering," Master's thesis, Lappeenranta University of Technology, 2004.

[63] T. Menzies, "Critical success metrics: Evaluation at the business-level," *International Journal of Human-Computer Studies, special issue on evaluation of KE techniques*, vol. 51, no. 4, pp. 783–799, October 1999, available from http://menzies.us/pdf/99csm.pdf.

[64] M. Harman and B. Jones, "Search-based software engineering," *Journal of Information and Software Technology*, vol. 43, pp. 833–839, December 2001.

[65] T. Mitchell, *Machine Learning*. McGraw-Hill, 1997.

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

[67] M. Druzdzel, "Some properties of joint probability distributions," in *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, 1994, pp. 187–194.

[68] A. L. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal*, vol. 3, no. 3, pp. 211–229, July 1959.

[69] R. Kohavi and G. H. John, "Wrappers for feature subset selection," *Artificial Intelligence*, vol. 97, no. 1-2, pp. 273–324, 1997. [Online]. Available: citeseer.nj.nec.com/kohavi96wrappers.html

[70] A. Avritzer, J. Ros, and E. Weyuker, "Reliability of rule-based systems," *IEEE Software*, pp. 76–82, September 1996.

[71] W. Wong and A. Mathur, "Reducing the cost of mutation testing: An empirical study," *The Journal of Systems and Software*, vol. 31, no. 3, pp. 185–196, December 1995.

[72] T. Budd, "Mutation analysis of programs test data," Ph.D. dissertation, Yale University, 1980.

[73] A. Acree, "On mutations," Ph.D. dissertation, School of Information and Computer Science, Georgia Institute of Technology, 1980.

[74] J. Bieman and J. Schultz, "An empirical evaluation (and specification) of the all-du-paths testing criterion," *Software Engineering Journal*, vol. 7, no. 1, pp. 43–51, 1992.

[75] M. Harrold, J. Jones, and G. Rothermel, "Empirical studies of control dependence graph size for c programs," *Empirical Software Engineering*, vol. 3, pp. 203–211, 1998.

[76] 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.

[77] G. J. Holzmann, "Algorithms for automated protocal verification," *ATT Technical Journal*, vol. 69, no. 2, pp. 32–44, 1990.

[78] G. Miller, "The magical number seven, plus or minus two: Some limits on our capacity for processing information," *The Psychological Review*, vol. 63, pp. 81–97, 1956, available from http://www.well.com/~smalin/miller.html.

[79] T. Menzies, "21st century ai: proud, not smug," *IEEE Intelligent Systems*, 2003, available from http://menzies.us/pdf/03aipride.pdf.

[80] S. Bay and M. 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.

[81] G. I. Webb, S. Butler, and D. Newlands, "On detecting differences between groups," in *KDD '03: Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. New York, NY, USA: ACM Press, 2003, pp. 256–265.

[82] C. Cai, A. Fu, C. Cheng, and 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.

[83] C. Blake and C. Merz, "UCI repository of machine learning databases," 1998, uRL: http://www.ics.uci.edu/~mlearn/MLRepository.html.

[84] D. 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.

[85] M. Fagan, "Advances in software inspections," *IEEE Trans. on Software Engineering*, pp. 744–751, July 1986.

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

[87] N. Leveson, S. Cha, and T. Shimall, "Safety verification of ADA programs using software fault trees," *IEEE Software*, vol. 8, no. 7, pp. 48–59, July 1991.

[88] A. Saltelli, K. Chan, and E. Scott, *Sensitivity Analysis*. Wiley, 2000.

[89] D. Boning and P. Mozumder, "Doe/opt: a system for design of experiments, response surface modeling, and optimization using process and device simulation," *IEEE Transactions on Semiconductor Manufacturing*, vol. 7, no. 2, pp. 233–244, May 1994.

[90] G. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

Tim Menzies is a associate professor at the Lane Department of Computer Science at West Virginia University (USA), and has been working with NASA on software quality issues since 1998. He has a CS degree and a PhD from the University of New South Wales and is the author of over 160 publications. His recent research concerns modeling and learning with a particular focus on light weight modeling methods. His doctoral research explored the validation of, possibly inconsistent, knowledge-based systems in the QMOD specification language.

James Kiper is Associate Dean for Research and Graduate Studies in the School of Engineering and Applied Science, and Professor of Computer Science at Miami University where he has been for the past twenty years. His research interest are in the area of software engineering, design rationale capture, representation, and analysis; and in software and system risk management.

Jeremy Greenwald is a graduate student in the Computer Science Department at Portland State University. He received his BS in Physics and Astronomy from the University of Pittsburgh in 2001. He has over six years of research experience in numeric methods and data mining. His master thesis focuses on combining data mining with numeric optimization techniques. He also has interned at a software development firm in Beaverton, Oregon.

Ying Hu is a software designer working in Vancouver, British Vancouver. Formerly a graduate student at School of Electrical Engineering at the University of British Columbia, Ms Hu implemented TAR2.

David Raffo is currently Associate Professor of Computer Science and Business Administration at Portland State University and a Principal Consultant at Quantel, Inc. Raffo completed his Ph.D. at Carnegie Mellon University. His research interests include: Strategic Process Design, Financial Analysis of Systems Engineering Decisions, Process Simulation, and Value Based Systems Engineering. Dr. Raffo has over forty refereed publications in the field of software engineering and is co-Editor-in-Chief of the international journal of Software Process: Improvement and Practice. He is a Visiting Scientist at the Software Engineering Institute and a Research Member of the International Process Research Consortium (IPRC). Prior professional experience includes programming as well as managing software development and consulting projects at Arthur D. Little, Inc. Dr. Raffo teaches courses in Software Process Improvement, Software Process Modeling and Simulation, and Systems Analysis and Design.

Siri-on Setamanit is currently a Consultant at Quantel, Inc., a Portland, Oregon based firm specializing in Process Simulation and Modeling tools and services. Ms. Setamanit's research interests include: Software Process Modeling and Simulation, Global Software Development processes, and Supply and Logistics Management. Ms. Setamanit received a M.S. and Ph.D. (2007) in Systems Science from Portland State University, an MBA from the University of Oregon, and a BBA from Chulalongkorn University in Bangkok, Thailand.

APPENDIX

Figure 12 and Figure 13 show the decision trees learned from the example of §V-B.

```
policy=NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF <= 0.5 :
|   policy=NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW, FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF <= 0.5 :
|  |   policy=WNNN,NWNN,FNNW,FNNB,BNNN,NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,
              NNWF,BBBB,FFFF,NNNF,BFFF <= 0.5 :
|  |  |   _detCap_8 <= 0.431 : 6940
|  |  |   _detCap_8 >  0.431 : 8180
|  |   policy=WNNN,NWNN,FNNW,FNNB,BNNN,NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,
              NNWF,BBBB,FFFF,NNNF,BFFF >  0.5 :
|  |  |   _detCap_8 <= 0.491 :
|  |  |  |   _inspDur_10 <= 94 : 8780
|  |  |  |   _inspDur_10 >  94 : 7860
|  |  |   _detCap_8 >  0.491 : 9320
|   policy=NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF >  0.5 :
|  |   policy=BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF <= 0.5 : 9750
|  |   policy=BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF >  0.5 : 10300
policy=NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF >  0.5 :
|   policy=NNWF,BBBB,FFFF,NNNF,BFFF <= 0.5 :
|  |   _detCap_38 <= 0.418 : 10700
|  |   _detCap_38 >  0.418 : 11400
|   policy=NNWF,BBBB,FFFF,NNNF,BFFF >  0.5 :
|  |   _corErr_50 <= 155 : 11900
|  |   _corErr_50 >  155 :
|  |  |   policy=NNNF,BFFF <= 0.5 : 12400
|  |  |   policy=NNNF,BFFF >  0.5 :
|  |  |  |   _detCap_38 <= 0.519 : 12500
|  |  |  |   _detCap_38 >  0.519 : 13600
```

Fig. 12. A regression tree learned by CART from data generated from the Figure 4 model. In this figure, and Figure 13, policies are denoted as a four-part sequence describing the inspection policy at each phase. For example, “FFFN” denotes using no inspections for code but full Fagan inspections for all earlier phases.

```
utility = 7370
 + 1220policy=WNNN,NWNN,FNNW,FNNB,BNNN,NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,
              NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 258policy=FNNB,BNNN,NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW, FNNF,
             NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 640policy=NNWN,NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 505policy=NNNW,WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 - 190policy=WNNW,BNNW,WNNB,BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 206policy=BNNB,WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 - 177policy=WWWW,NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 890policy=NNNB,NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 - 296policy=NNWW,FNNF,NNWB,NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 545policy=NNFN,BNNF,WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 640policy=WNNF,NNWF,BBBB,FFFF,NNNF,BFFF
 + 459policy=NNWF,BBBB,FFFF,NNNF,BFFF - 143policy=BBBB,FFFF,NNNF,BFFF
 + 835policy=NNNF,BFFF - 142_spec_2=n,b + 27_spec_2=b + 3690_detCap_8
 - 135_inspEff_9 - 7320_inspDur_10 - 64400_corErr_11 - 760_detCap_18
 - 134_inspEff_19 - 7310_inspDur_20 - 64300_corErr_21 - 9100_detCap_28
 - 134_inspEff_29 - 7320_inspDur_30 - 64300_corErr_31 + 6170_detCap_38
 - 134_inspEff_39 - 7320_inspDur_40 - 64400_corErr_41
+ 21500_inspEff_48 + 1.17e6_inspDur_49 + 64400_corErr_50
```

Fig. 13. Linear regression learned from data generated from the Figure 4 model.