2009 IEEE/ACM International Conference on Automated Software Engineering

Understanding the Value of Software Engineering Technologies

Phillip Green II, Tim Menzies, Steven Williams, Oussama El-Rawas
Lane Department of CS&EE, West Virginia University, USA,
deathcheese@yahoo.com, tim@menzies.us, stevencwilliams@gmail.com, orawas@gmail.com

***Abstract*—When AI search methods are applied to software process models, then appropriate technologies can be discovered for a software project. We show that those recommendations are greatly affected by the business context of its use. For example, the automatic defect reduction tools explored by the ASE community are only relevant to a subset of software projects, and only according to certain value criteria. Therefore, when arguing for the value of a particular technology, that argument should include a description of the value function of the target user community.**

***Keywords*-software economics; artificial intelligence**

I. Introduction

There are many software engineering (SE) *technologies* that a manager might apply in the hope of improving their software development project. Some of these technologies are *paper-based* methods like the checklists proposed by orthogonal defect classification [1]. Other technologies are *tool-based* such as using the new generation of functional programming languages or execution and testing tools [2] or automated formal analysis [3]. Yet other technologies are more *process-based* including process improvement initiatives, changing an organization's hiring practices, or a continual renegotiation of the requirements as part of an agile software development cycle [4].

SE technologies can be inappropriately applied to projects if a project manager do not assess the technology's *benefits* against its associated *drawbacks*. For example, using less-skilled developers is tempting (since there are so many of them), but the resulting product may be more defect-prone. As a result, products might get to market faster but contain too many defects.

In theory, software process models can be used to model the trade-offs associated with different technologies. However, such models can suffer from *tuning instability*. Large instabilities make it hard to recognize important influences on a project. For example, consider the following simplified COCOMO [5] model,

$$effort = a \cdot LOC^{b+pmat} \cdot acap \tag{1}$$

While simplified, the equation presents the core assumption of COCOMO; i.e. that software development effort is exponential on the size of the program. In this equation, (a, b) control the linear and exponential effects (respectively) on model estimates; while $pmat$ (process maturity) and $acap$ (analyst capability) are project choices adjusted by managers. Equation 1 contains two features $(acap, pmat)$ and a full COCOMO-II model contains 22 [5].

Baker [6] reports a study that learned values of (a, b) for a full COCOMO model using Boehm's local calibration method [7] from 30 randomly selected samples of 90% of the available project data. The ranges varied widely:

$$(2.2 \leq a \leq 9.18) \wedge (0.88 \leq b \leq 1.09) \tag{2}$$

Such large variations make it possible to misunderstand the effects of project options. Suppose some proposed technology doubles productivity, but a moves from 9 to 4.5. The improvement resulting from that change would be obscured by the tuning variance.

Previously we have reported some success with stochastic AI tools that search through the space of possible tunings within process models [8]–[12]. This approach returned conclusions about the project that were stable across the entire space of possibilities. For example, in two studies [8], [9] we found that automated defect removal tools (such as those discussed at the ASE conference) were often required to achieve minimum defects/ development effort/ development time[1]. Elsewhere [10], [11], we have used this tool to comparatively assess different proposed changes to a project.

Our prior reports did not compare the recommendations found by different AI search methods. Nor did we explore the effect of changing the $value$ function that models user goals. Here, we compare half a dozen AI methods and apply the best one to four case studies. Next, we repeat that analysis using a different $value$ function.

The results were very surprising. Prior to this research, we had a pre-experimental intuition that concepts of $value$ might change the organization of a project. However, we suspected that some things would remain constant (e.g., condoning the use of execution testing tools).

This turned out *not* to be the case. For the two value functions explored here, if one function approves of X then the other usually approves of $not\ X$. This result that *value can change everything* should motivate much future work on the business context of our tools.

[1] *Time* refers to calendar months required to complete a project while *effort* refers to the number of staff hours within those months. For example, 4 people working for one year takes $time = 12$ months and $effort = 48$ months.

II. Related Work

This work was inspired by Barry Boehm's 2004 ASE keynote [13] in which he advocated assessing SE tools by their value to a stake-holder, rather than just via their functionality. More specifically, one of the value functions used in this report was developed by Boehm and Huang [14].

In his description of value-based SE, Boehm favors continuous optimization methods to conduct cost/benefit trade-off studies. Tuning instability can confuse such continuous optimization methods. For example, the results of gradient descent methods can be misleading if the coefficients on the partial functions are very uncertain.

In our view, uncertainty in software process models is a very under-explored area. For example, the variance of Equation 2 remained undiscovered for 26 years until Baker, at our suggestion, looked for it. Much of the related work on uncertainty in software engineering uses a Bayesian analysis. For example, Pendharkar et al. [15] demonstrate the utility of Bayes networks in effort estimation while Fenton and Neil explore Bayes nets and defect prediction [16] (but unlike this paper, neither of these teams links defect models to effort models). We elect to take a non-Bayesian approach since most of the industrial and government contractors we work with use parametric models like COCOMO.

Other related work is the search-based SE approach advocated by Harman [17]. Search-Based Software Engineering (SBSE) uses optimization techniques from operations research and meta-heuristic search (e.g., simulated annealing and genetic algorithms) to hunt for near-optimal solutions to complex and over-constrained software engineering problems. Harman takes care to distinguish AI search-based methods from those seen in standard numeric optimizations. Such optimizers usually offer settings to all controllables. This may result in needlessly complex recommendations since a repeated empirical observation is that many model inputs are noisy or correlated in similar ways to model outputs [18]. Such noisy or correlated variables can be pruned away to generate simpler solutions that are easier and quicker to understand. In continuous domains, there is much work on feature selection [19] and techniques like principal component analysis [20] to reduce the number of dimensions reported by an analysis. Comparative studies report that discrete AI-based methods can do better at reducing the size of the reported theory [18].

The SBSE approach can and has been applied to many problems in software engineering (e.g., requirements engineering [21]) but most often in the field of software testing [2]. Harman's writing inspired us to try simulated annealing to search the what-ifs in untuned COCOMO models [9]. SA is a widely-used algorithm, perhaps due to the simplicity of its implementation and its very low memory requirements. Our results, shown below, indicate that several other algorithms out-perform SA (at least, on our models). This result strongly suggests that proponents of SA should try a broader range of search engines.

The paper compares SEESAW to five AI search algorithms. These half a dozen algorithms hardly represent an exhaustive list of possibilities. For example, Gu et al. [22] list hundreds of optimization algorithms and no single conference paper can experiment with them all. However, we make one comment as to why the above list does not include integer programming methods. Coarfa et al. [23] found that integer programming-based approaches ran an order of magnitude slower than discrete methods like SEESAW. Similar results were reported by Gu et al. where discrete methods ran 100 times faster than integer programming [22].

Our research grew out of a frustration with standard methods to reduce tuning variance. Previously, we have tried reducing that variance in various ways:

- Feature selection to prune spurious details [24];
- Instance selection to prune irrelevancies [25];
- Extended data collection.

Despite all that work, the variance observed in our models remains very large. Even the application of techniques such as instance-based learning have failed to reduce variance in our effort predictions [26]. Feature subset selection has also been disappointing: while it reduces the median performance variance somewhat (in our experiments, from 150% to 53% [25]), the residual error rates are large enough that it is hard to use the predictions of these models as evidence for the value of some proposed approach. Lastly, further data collection has not proven useful. Certainly, there is an increase in the availability of historical data on prior projects[2]. However, Kitchenham [27] cautions that the literature is contradictory regarding the value of using data from other companies to learn local models.

Having failed to tame tuning variance, despite years of research, we turned to alternate methods.

III. Dodging Tuning Variance

In order to tame prediction variance, we need to understand its source. The predictions of a model about a software engineering project are altered by project variables P and tuning variables T:

$$prediction = model(P, T) \tag{3}$$

For example, in Equation 1, the tuning options T are the range of (a, b) and the project options P are the range of $pmat$ (process maturity) and $acap$ (analyst capability). Based on the definitions of the COCOMO model we can say that the ranges of the project variables are $P = 1 \leq (pmat, acap) \leq 5$. Further, given the cone of uncertainty associated with a particular project p, we can identify the subset of the project options $p \subseteq P$ relevant to a particular project. For example, a project manager may be unsure of

[2]see http://promisedata.org.data/data and http://www.isbsg.org/.

the exact skill level of team members. However, if she were to assert "my analysts are better than most", then p would include $\{acap = 4, acap = 5\}$.

Our approach assumes that the dominant influences on the $prediction$ are the project options p (and not the tuning options T). Under this assumption, the predictions can be controlled by

- Constraining p (using some AI tool);
- While leaving T unconstrained (and sampling $t \in T$ using Monte Carlo methods).

Specifically, we seek a treatment $r_x \subseteq p$ that maximizes the $value$ of a model's predictions where *value* is a domain-specific function that scores model outputs according to user goals:

$$\arg\max_x \left(\overbrace{r_x \subseteq p}^{AI\ search}, \underbrace{t \subseteq T, value(model(r_x, t))}_{Monte\ Carlo} \right) \tag{4}$$

This approach is somewhat different from standard methods for decision support for software process models. From [28], we define standard practice as:

1) Collect domain knowledge.
2) Build an initial model based on step 1 including as yet unknown parameters. Note that these unknowns represent a range of tuning options.
3) Tune model by (e.g.) regression on local data.
4) Conduct sensitivity analysis on the tuned models.

These four steps can take quite some time and, at least in our experience with more complex versions of Equation 1, may result in models with large tuning instabilities. The sensitivity analysis, if conducted using gradient descent methods, will not be successful for our models since the gradients exhibit the very large variances of Equation 2. Hence, another method that need not wait for time-consuming (and possibly pointless) local data collection and tuning:

1) Check the literature for software process models where the dominant influences on predictions are the project options P, not the tuning options T.
2) Map project options to model input options.
3) Sample those models using AI tools to constrain the project options, while sampling the tuning options with Monte Carlo methods (i.e., Equation 4).

One heuristic for checking the literature in step 1 is to avoid overly elaborate models whose authors may have extended the model far beyond what can be supported with the available data. We suspect a "Goldilocks" principle might be appropriate:

- Tiny models offer trite conclusions and are insensitive to important project features.
- Very large models may need much data collection to constrain the tunings.
- In between there may exist some models that are "just right"; i.e., big enough to draw interesting conclusions, but small enough such that the internal tuning variance does not dominate the variance results from input project options.

scale factors (exponentially decrease effort&cost)	prec:	have we done this before?
	flex:	development flexibility
	resl:	any risk resolution activities?
	team:	team cohesion
	pmat:	process maturity
upper (linearly decrease effort&cost)	acap:	analyst capability
	pcap:	programmer capability
	pcon:	programmer continuity
	aexp:	analyst experience
	pexp:	programmer experience
	ltex:	language and tool experience
	tool:	tool use
	site:	multiple site development
	sced:	length of schedule
lower (linearly increase effort&cost)	rely:	required reliability
	data:	secondary memory storage requirements
	cplx:	program complexity
	ruse:	software reuse
	docu:	documentation requirements
	time:	runtime pressure
	stor:	main memory requirements
	pvol:	platform volatility

Figure 1. The COCOMO "scale factors" and "effort multipliers" change effort and cost by an exponential and linear amount (respectively). Increasing these values has the effect described in column one.

aa:	automated analysis
etat:	execution-based testing and tools
pr:	peer-reviews

Figure 2. The COQUALMO defect removal methods. Increasing these values decreases delivered defects.

We make no claim that *all* process models are "just right" and, hence, can be controlled via Equation 4. Some process models can be quite complex and include: discrete-event models [29], [30]; system dynamics models [31]; state-based models [32]–[34]; rule-based programs [35]; or standard programming constructs such as those used in Little-JIL [36], [37]. These rich modeling frameworks allow the representation of detailed insights into an organization. However, the effort required to tune them is non-trivial.

In terms of the Goldilocks principle, we suspect that many process models may not be near the "right size" and will require extensive tuning before they can be used for decision making. Fortunately, we have found that the USC COCOMO and COQUALMO models [38] are "just right". In all our studies [8]–[12] we have found that prediction variance can be controlled by only constraining project options while letting the tuning variance remaining unchecked. Hence, we use these models for our research. One advantage of these models is that they are fully described in the literature. The same can not be said for other commercial models such as PRICE TRUE PLANNING [39], SLIM [40], or SEER-SEM [41]. Also, at least for the COCOMO effort model, there exist baseline results [42].

IV. Model Details

COCOMO offers effort and time predictions while COQUALMO offers defect predictions. Using the models we can represent the project options P and tuning options T of Equation 3 as follows.

A. Project Options: P

COCOMO and COQUALMO's features are shown in Figure 1 and Figure 2. The features have a range taken from {very low, low, nominal, high, very high, extremely high} or

$$\{vl = 1, l = 2, n = 3, h = 4, vh = 5, xh = 6\}$$

These features include manual methods for defect removal. High values for peer reviews (or *pr*, see Figure 2) denote formal peer group review activities (participants have well defined and separate roles, the reviews are guided by extensive review checklists/root cause analysis, and reviews are a continuous process guided by statistical control theory [43]).

COQUALMO also models automatic methods for defect removal. Chulani [44] defines the top half of *automated analysis* as:

4 (high): intermediate-level module and inter-module code syntax and semantic analysis. Simple requirements/design view consistency checking.

5 (very high): More elaborate requirements/design view consistency checking. Basic distributed-processing and temporal analysis, model checking, symbolic execution.

6 (extremely high): Formalized[3] specification and verification. Temporal analysis, model checking, symbolic execution.

The top half of *execution-based testing and tools* is:

4 (high): Well-defined test sequence tailored to organization (acceptance / alpha / beta / flight / etc.) test. Basic test coverage tools, test support system.

5 (very high): More advanced tools, test data preparation, basic test oracle support, distributed monitoring and analysis, assertion checking. Metrics-based test process management.

6 (extremely high): Highly advanced tools: oracles, distributed monitoring and analysis, assertion checking. Integration of automated analysis and test tools. Model-based test process management.

In the sequel, the following observation will become important: Figure 1 is much longer than Figure 2. This reflects a modeling intuition of COCOMO/COQUALMO: it is better to prevent the introduction of defects (using changes to Figure 1) than to try and find them, once they have been introduced (using Figure 2).

[3]Consistency-checkable pre- conditions and post-conditions, but not necessarily mathematical theorems.

B. Tuning Options: T

For COCOMO effort multipliers (the features that that affect effort/cost in a linear manner), the off-nominal ranges {vl=1, l=2, h=4, vh=5, xh=6} change the prediction by some ratio. The nominal range {n=3}, however, corresponds to an effort multiplier of 1, causing no change to the prediction. Hence, these ranges can be modeled as straight lines $y = mx + b$ passing through the point (x, y)=$(3, 1)$. Such a line has a y-intercept of $b = 1 - 3m$. Substituting this value of b into $y = mx + b$ yields:

$$\forall x \in \{1..6\}\ EM_i = m_\alpha(x - 3) + 1 \qquad (5)$$

where m_α is the effect of α on effort/cost.

We can also derive a general equation for the features that influence cost/effort in an exponential manner. These features do not "hinge" around (3,1) but take the following form:

$$\forall x \in \{1..6\}\ SF_i = m_\beta(x - 6) \qquad (6)$$

where m_β is the effect of factor i on effort/cost.

COQUALMO contains equations of the same syntactic form as Equation 5 and Equation 6, but with different coefficients. Using experience for 161 projects [5], we can find the maximum and minimum values ever assigned to m for COQUALMO and COCOMO. Hence, to explore tuning variance (the $t \in T$ term in Equation 4), all we need to do is select m values at random from the min/max m values ever seen. An appendix to this document lists those ranges.

C. Case Studies: $p \subseteq P$

We use p to denote the subset of the project options $p_i \subseteq P$ relevant to particular projects. The four particular projects p_1, p_2, p_3, p_4 used as the case studies of this paper are shown in in Figure 3:

- OSP is the GNC (guidance, navigation, and control) component of NASA's *Orbital Space Plane*;
- OSP2 is a later version of OSP;
- Flight and ground systems reflect typical ranges seen at NASA's Jet Propulsion Laboratory.

Some of the features in Figure 3 are known precisely (see all the features with single *fixed settings*). But many of the features in Figure 3 do not have precise settings (see all the features that *range* from some *low* to *high* value). Sometimes the ranges are very narrow (e.g., the process maturity of JPL ground software is between 2 and 3), and sometimes the ranges are very broad.

Figure 3 does not mention all the features listed in Figure 1 inputs. For example, our defect predictor has inputs for use of *automated analysis*, *peer reviews*, and *execution-based testing tools*. For all inputs not mentioned in Figure 3, ranges are picked at random from (usually) {1, 2, 3, 4, 5}.

	ranges			fixed settings	
project p_i	feature	low	high	feature	setting
	prec	1	2	data	3
p_1=OSP: Orbital space plane	flex	2	5	pvol	2
	resl	1	3	rely	5
	team	2	3	pcap	3
	pmat	1	4	plex	3
	stor	3	5	site	3
	ruse	2	4		
	docu	2	4		
	acap	2	3		
	pcon	2	3		
	apex	2	3		
	ltex	2	4		
	tool	2	3		
	sced	1	3		
	cplx	5	6		
	KSLOC	75	125		
	prec	3	5	flex	3
p_2=OSP2	pmat	4	5	resl	4
	docu	3	4	team	3
	ltex	2	5	time	3
	sced	2	4	stor	3
	KSLOC	75	125	data	4
				pvol	3
				ruse	4
				rely	5
				acap	4
				pcap	3
				pcon	3
				apex	4
				plex	4
				tool	5
				cplx	4
				site	6
	rely	3	5	tool	2
p_3=JPL flight software	data	2	3	sced	3
	cplx	3	6		
	time	3	4		
	stor	3	4		
	acap	3	5		
	apex	2	5		
	pcap	3	5		
	plex	1	4		
	ltex	1	4		
	pmat	2	3		
	KSLOC	7	418		
	rely	1	4	tool	2
p_4=JPL ground software	data	2	3	sced	3
	cplx	1	4		
	time	3	4		
	stor	3	4		
	acap	3	5		
	apex	2	5		
	pcap	3	5		
	plex	1	4		
	ltex	1	4		
	pmat	2	3		
	KSLOC	11	392		

Figure 3. Four case studies. Numeric values $\{1, 2, 3, 4, 5, 6\}$ map to *very low, low, nominal, high, very high, extra high*. This data comes from experienced NASA managers summarizing over real-world projects.

D. Value

If there exist multiple possible treatments, a $value$ function is required to rank alternatives. The $value$ function should model the goals of the business users who are making project decisions about some software development. At the end of this paper, we compare results from two very different $value$ functions.

1) "BFC" = Better, Faster, Cheaper: Ideally, software is built with fewer defects D, using less effort E, and in shorter time T. A value function for this goal can be modeled as the Euclidean distance to minimum effort, time, defects:

$$bfc = \sqrt{f\bar{T}^2 + c\bar{E}^2 + \left(b\bar{D}\left(1 + 1.8^{rely-3}\right)\right)^2} \quad (7)$$

$$value_{bfc} = \frac{1}{bfc} \quad (8)$$

In the above, $value$ is highest when defects and effort and development time are lowest. Also, $0 \leq (b, f, c) \leq 1$ represents the business importance of (better, faster, cheaper). For this study, we use $b = f = c = 1$. In other work, we have explored the effects of using other b, f, c values [12].

In Equation 7, $\bar{T}, \bar{E}, \bar{D}$ are the time, effort, and defect scores normalized zero to one. Equation 7 models the business intuition that defects in high reliability systems and exponentially more troublesome than in low reliability systems:

- If reliability moves from very low to very hight (1 to 6), the term 1.8^{rely-3} models a function that (a) is ten times larger for very high than very low reliability systems; and (b) passes through 1 at $rely = 3$ (so systems with nominal reliability do not change the importance of defects).

2) "XPOS" = Risk Exposure: The BFC value function is somewhat idealistic in that it seeks to remove *all* defects by spending *less* money on *faster* developments. An alternate value function comes from Huang and Boehm [14]. This alternate value function, which we call "XPOS", models the situation where a software company must rush a product to market, without compromising too much on software quality. Based on Huang's Ph.D. dissertation [45], we operationalize XPOS as follows.

Huang defines business risk exposure (RE) as a combination of *software quality investment risk exposure* (RE_q) and *market share erosion risk exposure* (RE_m). We invert that expression to yield $value_{XPOS}$ (so an exposed project has low value):

$$RE = RE_q + RE_m \quad (9)$$

$$value_{XPOS} = \frac{1}{RE} \quad (10)$$

RE_q values high-quality software and therefore prioritizes quality over time. RE_q is composed of two primary components: probability of loss due to unacceptable quality $P_q(L)$ and size of loss due to unacceptable quality $S_q(L)$. $P_q(L)$ is calculated based on defects. $S_q(L)$ is calculated based on complexity (the COCOMO $cplx$ feature), reliability ($rely$), and a cost function. S_c is a value from a Pareto-valued table based on $rely$. We choose the project months estimate as the basis of this cost function.

$$RE_q = P_q(L) * S_q(L) \quad (11)$$

$$P_q(L) = \frac{defects}{defects_{vl}} \quad (12)$$

$$S_q(L) = 3^{\frac{cplx-3}{2}} \cdot PM \cdot S_c \quad (13)$$

In Equation 12, $defects_{vl}$ is the lower bound on defects for that project.

In Equation 13 the $\frac{cplx-3}{2}$ term is similar to the $\bar{D}$ coefficient inside Equation 7: if complexity changes below or above 3, then it reduces or adds (respectively) to the unacceptable quality risk. However at $cplx = 3$, the multiplier is one (i.e., no effect).

RE_m values a fast time-to-market and therefore prioritizes time over quality. RE_m is calculated from PM and reliability ($rely$). M_c is a value from a exponential-valued table based on $rely$.

$$RE_m = PM \cdot M_c \tag{14}$$

V. Searching for r_x

Our search runs two phases: a *forward select* and a *back select* phase. The *forward select* grows r_x, starting with the empty set. At each round i in the forward select one or more ranges (e.g., $acap = 3$) are added to r_x. The resulting r_x set found at round i is denoted r_x^i.

The forward select ends when the search engine cannot find more ranges to usefully add to r_x^i. Before termination, we say that the *open* features at round i are the features in Figure 1 and Figure 2 not mentioned by any range in r_x^i. The value of r_x^i is assessed by running the model N times with:

1) all of r_x^i
2) any $t \in T$, selected at random
3) any range at random for *open* features.

In order to ensure minimality, a *back select* checks if the final r_x set can be pruned. If the forward select caches the simulation results seen at each round i, the back select can perform statistical tests to see if the results of round $i - 1$ are significantly different from round i. If the difference is *not* statistically significant, then the ranges added at round i are *pruned* away and the back select recurses for $i - 1$. We call the un-pruned ranges the *selected* ranges and the point where pruning stops the *policy point*.

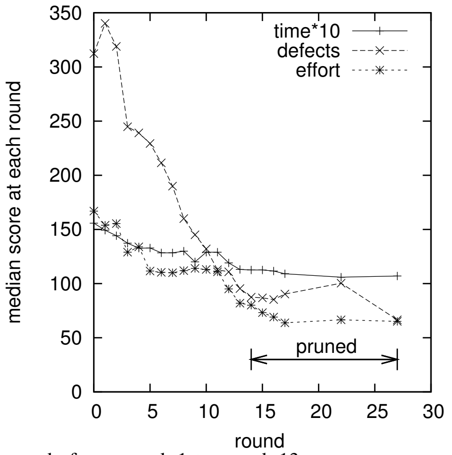
For example, in Figure 4, the policy point is round 13 and the decisions made at subsequent rounds are pruned by the back select. That is, the treatments returned by our search engines are all the ranges r_x^i for $1 \leq i \leq 13$. The *selected* ranges are shown in a table at the bottom of the figure and the effects of applying the conjunction of ranges in r_x^{13} can be seen by comparing the values at round=0 to round=13:

- Defects/KLOC reduced: 350 to 75;
- Time reduced: 16 to 10 months;
- Effort reduced: 170 to 80 staff months.

A. Alternate Search Methods

This paper implements the forward select using SA, MaxWalkSat, SEESAW, BEAM, A-STAR and ISAMP.

SA: In our initial experiments [9], we ran forward/back selects as a post-processor to a simulated annealer. Consider a typical SA run that has explored 10,000 variants on some solution. A side-effect of that run is 10,000 sets of inputs, each scored with a value function. Our tool "STAR" classified the outputs into 10% *best* values and 90% *rest*. All the ranges from all the features were then sorted into



Decisions made from round=1 to round=13:

round0: $r_x = \emptyset$	round7: added {rely=3}
round1 added {pmat=3}	round8: added {stor = 3}
round2: added {resl=4}	round9: added {time = 3}
round3: added {team=5}	round10: added {tool = 4}
round4: added {aexp=4}	round11: added {sced = 2}
round5: added {docu=3}	round12: added {site = 4}
round6: added {plex=4}	round13: added {acap = 5}

Figure 4. Example forward and back select results.

a list of length R according to how much more frequently they appeared in *best* than *rest*. The forward select was then called using the first i items $1 \leq i \leq R$.

MaxWalkSat is a local search algorithm [46]. Given a random selected treatment, MaxWalkSat attempts n modifications to randomly selected features. Sometimes (controlled by the α parameter), the modification is "smart": for the selected feature, the algorithm chooses the range that minimizes the value of the current solution. The rest of the time (i.e., at probability $1-\alpha$), a random range is chosen for the feature. Occasionally, MaxWalkSat will reset to a new randomly selected initial solution and, after N attempts, the best solution is returned. Our implementation used $n = 50$, $\alpha = 0.5$, and $N = 10$.

SEESAW is a variant of MaxWalkSat we first reported in [12]. While searching the ranges of a feature, this algorithm exploits the monotonic nature of Equation 5 and Equation 6. SEESAW ignores all ranges except the minimum and maximum values for a feature in p. Like MaxWalkSat, the feature chosen on each iteration is made randomly. However, SEESAW has the ability to delay bad decisions until the end of the algorithm (i.e., decisions where constraining the feature to *either* the minimum or maximum value results in a worse solution). These treatments are then guaranteed to be pruned during the back select.

ISAMP is a fast stochastic iterative sampling method that extends a treatment using randomly selected ranges. The algorithm follows one solution, then resets to try other paths (our implementation resets 20 times). The algorithm has proved remarkably effective at scheduling problems, perhaps because it can rapidly explore more of the search space [47]. To avoid exploring low-value regions, our version of ISAMP stores the worst solution seen so far. Any conjunction whose $value$ exceeds that of the worst solution is abandoned, and the new "worst value" is stored. If a conjunction runs out of new ranges to add, then the "worst value" is slightly decreased. This ensures that consecutive failing searches do not permanently raise the "worst value" by an overly permissive amount.

Our remaining algorithms use some variant of tree search. Each branch of the tree is a different treatment (a conjunction of ranges) of size i. In terms of the forward select search described above, the y-axis statistics of Figure 4 are collected whenever branches of size i are extended to size $i + 1$.

BEAM search extends branches as follows. Each branch forks once for every new option available to that range. All the new leaves are sorted by their value and only the top N ranked branches are marked for further expansion. For this study we used $N = 10$ and the y axis of Figure 4 was reported using the median values seen in the top N branches.

A-STAR runs like BEAM, but the sort order is determined by the sum f (the cost of reaching the current solution) plus g (a heuristic estimate of the cost to reach the final solution). Also, unlike BEAM, the list of options is not truncated so a termination criterion is needed (we stop the search if the best solution so far has not improved after m iterations). For this study, we used Equation 7 for g and the percentage of the features with ranges in the current branch as f.

VI. Experiments

A. Comparing Algorithms

We ran 20 forward selects on the case studies of Figure 3 using BFC followed by back selects (t-tests pruned round i if it was statistically the same at 95% confidence as round $i - 1$). Separate statistics were collected for the defects/effort/time predictions seen at the policy point in the 20*4 trials. The *top-ranked* algorithm(s) had statistically different and lower defects/effort/time predictions than any other algorithm(s).

Figure 5 shows how many times each algorithm was *top-ranked*. Note that the maximum value possible is 4 (i.e., once for each Figure 3 case study). In those results, the two stand-out worst algorithms are MaxWalkSat and ISAMP and the two stand-out best algorithms are SEESAW and BEAM.

algorithm	Defects	months	time
SEESAW	4	4	3
BEAM	0	3	3
A-star	0	1	1
SA	0	1	1
MaxWalkSat	0	0	0
ISSAMP	0	0	0

Figure 5. Number of times algorithm found to be top-ranked from 20 repeats of forward/back selecting over the four case studies of Figure 3.

data set	defects	time	effort
flight	80%	39%	72%
ground	85%	38%	73%
osp	65%	4%	42%
ops2	26%	22%	5%
median	73%	30%	57%

Figure 6. Percent reductions $(1 - final/initial)$ achieved by SEESAW on the Figure 3 case studies. The $initial$ values come from round 0 of the forward select. The $final$ values come from the policy point. Note that all the initial and final values are statistically different (Mann-Whitney, 95% confidence).

The performance of these two is sometimes equivalent (e.g., in $time$, both algorithms achieved an equal number of top ranks). However, we cannot recommend BEAM search for this domain:

- BEAM runs 10 times slower than SEESAW.
- SEESAW performs better than BEAM in some cases (e.g. in defects, BEAM is never top-ranked).

Figure 5 shows a great difference between MaxWalkSat and SEESAW results. As mentioned above, the difference between these two algorithms is very small: SEESAW assumed that the local search state space was monotonic, so it only explored minimum and maximum values for each feature. This single domain heuristic resulted in a dramatic performance improvement. This is not a new result: AI research in the 1980s [48] showed the value of combining domain-general "weak" heuristic algorithms (e.g., A-STAR, BEAM search) with domain-specific "strong" heuristics such as the monotonicity assumption exploited by SEESAW. However, even if it is not a new finding, it should remind researchers not to depend too heavily on off-the-shelf algorithms. As shown here, a little algorithm tweaking with domain knowledge can go a long way.

Figure 6 shows the percent reductions achieved by SEESAW on the four case studies of Figure 3. SEESAW's treatments can reduce project defect/time/effort estimates by 73, 30, and 57% (respectively). That is, even if SEESAW cannot offer a major reduction in the development time, it can advise how to reduce defects and development effort by over half. Based on Figure 6, we would recommend using SEESAW before all the project decisions are made, otherwise there is very little for it to work with. For example, SEESAW's worst performance was with the OSP2 case study. As shown in Figure 3, this project has the most amount of fixed settings, so SEESAW only has a small range of options it can explore.

B. Effects of Changing the Value Function

Our prior ASE report on this work [49] used simulated annealing to find treatments. That work reported that, in all studied cases, automated defect removal tools were always found in the learned treatments.

The poor performance of SA in Figure 6 means we must revisit those conclusions, using SEESAW rather than SA.

Data	Range	$value$		$\frac{B}{B+X}$	defect removal	
		B=BFC	X=XPOS		manual	automatic
ground	rely = 4	70	20	77		
	aa = 6	70	25	73		hi in B
	resl = 6	65	40	61		
	etat = 1	35	65	35		lo in X
	aexp = 5	45	85	34		
	pr = 1	35	80	30	lo in X	
	aa = 1	25	60	29		lo in X
	data = 2	25	70	26		
	rely = 1	15	70	17		
flight	rely = 5	65	25	72		
	flex = 6	80	50	61		
	docu = 1	55	85	39		
	site = 6	55	85	39		
	resl = 6	45	70	39		
	pr = 1	45	70	39	lo in X	
	pvol = 2	45	75	37		
	data = 2	35	60	36		
	cplx = 3	45	90	33		
	rely = 3	15	60	20		
OSP	pmat = 4	85	45	65		
	resl = 3	45	70	39		
	ruse = 2	40	65	38		
	docu = 2	25	90	21		
OSP2	sced = 2	100	0	100		
	sced = 4	0	80	0		

Figure 7. Frequency (in percents) of feature ranges seen in 20 repeats of SEESAW, using two different goal functions: BFC and XPOS. The last two columns comment on any defect reduction feature. Not shown in this figure are any feature ranges that occur less than 50% of the time.

Figure 7 shows the ranges seen in SEESAW's treatment (after a back select). The BFC and XPOS columns show the percent frequency of a range appearing when SEESAW used our different $value$ functions. These experiments were repeated 20 times and only the ranges found in the majority (more than 50%) of the trials are reported.

The results are divided into our four case studies: ground, flight, OSP, and OSP2. Within each case study, the results are sorted by the fraction $\frac{BFC}{BFC+XPOS}$. This fraction ranges 0 to 100 and:

- If close to 100, then a range is selected by BFC more than XPOS.
- If close to 0, then a range is selected by XPOS more than BFC.

The right-hand columns of Figure 7 flag the presence of manual defect remove methods (pr=peer reviews) or automatic defect removal methods (aa=automated analysis; $etat$=execution testing tools). Note that high levels of automatic defect removal methods are only frequently required in ground systems, and only when valuing BFC. More usually, defect removal techniques are *not* recommended. In ground systems, $etat = 1$, $pr = 1$, and $aa = 1$ are all examples of SEESAW *discouraging* rather than endorsing the use defect removal methods. That is, in three of our four case studies, it is more important to prevent defect introduction than to use after-the-fact defect removal methods. In ground, OSP, and OSP2, defect removal methods are very rare (only $pr = 1$ in flight systems).

Another important aspect of Figure 7 is that there is no example of both value functions frequently endorsing the same range. If a range is commonly selected by BFC, then it is usually *not* commonly accepted by XPOS. The most dramatic example of this is the OSP2 results of Figure 7: BFC always selects (at 100%) the low end of a feature ($sced$=2) while XPOS nearly always selects (at 80% frequency) the opposite high end of that feature.

In summary, perceptions of $value$ can change everything. Techniques that seem useful to one kind of project/value function may be counter-indicated for another. One characterization of the Figure 7 results is that, for some projects, it is preferable to prevent defects before they arrive (by reorganizing the project) rather than try to remove them afterwards using (say) peer review, automated analysis, or execution test tools.

VII. Conclusions

There are many different SE technologies that can be applied to a project. Such technologies can be defined as conjunctions of inputs to software process models. Technology X is preferred to technology Y if X's inputs lead to better output scores that Y.

Assessing the value of different SE technologies via process models is complicated by tuning instabilities. For several years, we have tried to reduce this problem using a variety of techniques. The failure of that work has prompted research on alternate methods. We dodge the model tuning problem by searching for conclusions that are stable across the space of possible tunings. Our tool uses AI to search for different project options. Each option is scored by a Monte Carlo run that selects tuning options at random from the space of possible options. For models where project values (and not tuning values) are the dominant influence on predictions, this combined method finds critical project options without requiring local tuning data. This is a very different approach to standard software process modeling where local data is essential to model tuning.

This paper certified that our current preferred search engine (SEESAW) runs as well (or better) than several alternatives. When we applied SEESAW with two different $value$ functions, the treatments found by SEESAW changed dramatically. In particular, most projects worked best when they were organized to stop defects entering the code base rather than using some automated analysis method to remove them, after they have been introduced.

This work has significant implications for the ASE community. Based on this work we argue that it is no longer enough to just propose (say) some automated defect reduction tool. Rather, the value of ASE tools for a software project needs to be carefully assessed with respect to the core $value$s of that project.

References

[1] R. Chillarege, I. S. Bhandari, J. K. Chaar, M. J. Halliday, D. S. Moebus, B. K. Ray, and M.-Y. Wong, "Orthogonal defect classification–a concept for in-process measurements," *IEEE Transactions on Software Engineering*, vol. 18, no. 11, pp. 943–956, November 1992.

[2] J. Andrews, F. Li, and T. Menzies, "Nighthawk: A two-level genetic-random unit test data generator," in *IEEE ASE'07*, 2007, available from http://menzies.us/pdf/07ase-nighthawk.pdf.

[3] G. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[4] D. Port, A. Olkov, and T. Menzies, "Using simulation to investigate requirements prioritization strategies," in *IEEE ASE'08*, 2008, available from http://menzies.us/pdf/08simrequire.pdf.

[5] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[6] D. Baker, "A hybrid approach to expert and model-based effort estimation," Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University, 2007, available from https://eidr.wvu.edu/etd/documentdata.eTD?documentid=5443.

[7] B. Boehm, *Software Engineering Economics*. Prentice Hall, 1981.

[8] T. Menzies, O. Elrawas, D. Baker, J. Hihn, and K. Lum, "On the value of stochastic abduction (if you fix everything, you lose fixes for everything else)," in *International Workshop on Living with Uncertainty (an ASE'07 co-located event)*, 2007, available from http://menzies.us/pdf/07fix.pdf.

[9] T. Menzies, O. Elrawas, J. Hihn, M. Feathear, B. Boehm, and R. Madachy, "The business case for automated software engineerng," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 303–312, available from http://menzies.us/pdf/07casease-v0.pdf.

[10] T. Menzies, S. Williams, O. El-waras, B. Boehm, and J. Hihn, "How to avoid drastic software process change (using stochastic statbility)," in *ICSE'09*, 2009, available from http://menzies.us/pdf/08drastic.pdf.

[11] T. Menzies, S. Williams, O. Elrawas, D. Baker, B. Boehm, J. Hihn, K. Lum, and R. Madachy, "Accurate estimates without local data?" *Software Process Improvement and Practice (to appear)*, 2009.

[12] T. Menzies, O. El-Rawas, J. Hihn, and B. Boehm, "Can we build software faster and better and cheaper?" in *PROMISE'09*, 2009, available from http://menzies.us/pdf/09bfc.pdf.

[13] B. Boehm, "Automating value-based software engineering, Keynote address to IEEE ASE," 2004, available from http://ase-conferences.org/ase/past/ase2004/download/KeynoteBoehm.pdf.

[14] L. Huang and B. Boehm, "How much software quality investment is enough: A value-based approach," *Software, IEEE*, vol. 23, no. 5, pp. 88–95, Sept.-Oct. 2006.

[15] P. C. Pendharkar, G. H. Subramanian, and J. A. Rodger, "A probabilistic model for predicting software development effort," *IEEE Trans. Softw. Eng.*, vol. 31, no. 7, pp. 615–624, 2005.

[16] N. E. Fenton and M. Neil, "A critique of software defect prediction models," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 675–689, 1999, available from http://citeseer.nj.nec.com/fenton99critique.html.

[17] M. Harman and J. Wegener, "Getting results from search-based approaches to software engineering," in *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 728–729.

[18] M. Hall and G. Holmes, "Benchmarking attribute selection techniques for discrete class data mining," *IEEE Transactions On Knowledge And Data Engineering*, vol. 15, no. 6, pp. 1437– 1447, 2003, available from http://www.cs.waikato.ac.nz/~mhall/HallHolmesTKDE.pdf.

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

[20] W. Dillon and M. Goldstein, *Multivariate Analysis: Methods and Applications*. Wiley-Interscience, 1984.

[21] O. Jalali, T. Menzies, and M. Feather, "Optimizing requirements decisions with keys," in *Proceedings of the PROMISE 2008 Workshop (ICSE)*, 2008, available from http://menzies.us/pdf/08keys.pdf.

[22] J. Gu, P. W. Purdom, J. Franco, and B. W. Wah, "Algorithms for the satisfiability (sat) problem: A survey," in *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1997, pp. 19–152.

[23] C. Coarfa, D. D. Demopoulos, A. San, M. Aguirre, D. Subramanian, and M. Y. Vardi, "Random 3-sat: The plot thickens," in *In Principles and Practice of Constraint Programming*, 2000, pp. 143–159, availabe from http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3662.

[24] Z. Chen, T. Menzies, and D. Port, "Feature subset selection can improve software cost estimation," in *PROMISE'05*, 2005, available from http://menzies.us/pdf/05/fsscocomo.pdf.

[25] T. Menzies, Z. Chen, J. Hihn, and K. Lum, "Selecting best practices for effort estimation," *IEEE Transactions on Software Engineering*, November 2006, available from http://menzies.us/pdf/06coseekmo.pdf.

[26] O. Jalali, "Evaluation bias in effort estimation," Master's thesis, Lane Department of Computer Science and Electrical Engineering, West Virginia University, 2007.

[27] B. A. Kitchenham, E. Mendes, and G. H. Travassos, "Cross- vs. within-company cost estimation studies: A systematic review," *IEEE Transactions on Software Engineering*, pp. 316–329, May 2007.

[28] J. Hihn, T. Menzies, K. Lum, T. Menzies, D. Baker, and O. Jalali, "2CEE, a Twenty First Century Effort Estimation Methodology," in *ISPA'08: International Society of Parametric Analysis*, 2008, available from http://menzies.us/pdf/08ispa.pdf.

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

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

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] A. Cass, B. S. Lerner, E. McCall, L. Osterweil, S. M. S. Jr., and A. Wise, "Little-jil/juliette: A process definition language and interpreter," in *Proceedings of the 22nd International Conference on Software Engineering (ICSE 2000)*, June 2000, pp. 754–757.

[37] A. Wise, A. Cass, B. S. Lerner, E. McCall, L. Osterweil, and J. S.M. Sutton, "Using little-jil to coordinate agents in software engineering," in *Proceedings of the Automated Software Engineering Conference (ASE 2000) Grenoble, France.*, September 2000, available from ftp://ftp.cs.umass.edu/pub/techrept/techreport/2000/UM-CS-2000-045.ps.

[38] B. Boehm, "Safe and simple software cost analysis," *IEEE Software*, pp. 14–17, September/October 2000, available from http://www.computer.org/certification/beta/Boehm_Safe.pdf.

[39] R. Park, "The central equations of the price software cost model," in *4th COCOMO Users Group Meeting*, November 1988.

[40] L. Putnam and W. Myers, *Measures for Excellence*. Yourdon Press Computing Series, 1992.

[41] R. Jensen, "An improved macrolevel software development resource estimation model," in *5th ISPA Conference*, April 1983, pp. 88–92.

[42] S. Chulani, B. Boehm, and B. Steece, "Bayesian analysis of empirical software engineering cost models," *IEEE Transaction on Software Engineerining*, vol. 25, no. 4, July/August 1999.

[43] F. Shull, F. Lanubile, and V. Basili, "Investigating reading techniques for object-oriented framework learning," *IEEE Transactions of Software Engineering*, vol. 26, no. 11, pp. 1101–1118, 2000.

[44] S. Devnani-Chulani, "Bayesian analysis of software cost and quality models," Ph.D. dissertation, 1999, available on-line at http://citeseer.ist.psu.edu/devnani-chulani99bayesian.html.

[45] L. Huang, "Software quality analysis: A value-based approach," Ph.D. dissertation, Department of Computer Science, University of Southern California, 2006, available from http://csse.usc.edu/csse/TECHRPTS/PhD_Dissertations/files/Huang_Dissert%ation.pdf.

[46] B. Selman, H. A. Kautz, and B. Cohen, "Local search strategies for satisfiability testing," in *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, M. Trick and D. S. Johnson, Eds., Providence RI, 1993. [Online]. Available: citeseer.ist.psu.edu/selman95local.html

[47] S. Craw, D. Sleeman, R. Boswell, and L. Carbonara, "Is knowledge refinement different from theory revision?" in *Proceedings of the MLNet Familiarization Workshop on Theory Revision and Restructuring in Machine Learning (ECML-94)*, S. Wrobel, Ed., 1994, pp. 32–34.

[48] T. McCluskey, "Combining weak learning heuristics in general problem solvers," in *IJCAI'87*, 1987, pp. 331–333.

[49] T. Menzies, D.Owen, and J. Richardson, "The strangest thing about software," *IEEE Computer*, 2007, http://menzies.us/pdf/07strange.pdf.

Appendix

This appendix lists the minimum and maximum m values used for Equation 5 and Equation 6. In the following, m_α and m_β denote COCOMO's linear and exponential influences on effort/cost, and m_γ and m_δ denote COQUALMO's linear and exponential influences on number of defects.

Their are two sets of effort/cost multipliers:

1) The *positive effort EM* features, with slopes m_α^+, that are proportional to effort/cost. These features are: cplx, data, docu, pvol, rely, ruse, stor, and time.
2) The *negative effort EM* features, with slopes m_α^-, are inversely proportional to effort/cost. These features are acap, apex, ltex, pcap, pcon, plex, sced, site, and tool.

Their m ranges, as seen in 161 projects [38], are:

$$\left(0.073 \leq m_\alpha^+ \leq 0.21\right) \wedge \left(-0.178 \leq m_\alpha^- \leq -0.078\right) \tag{15}$$

In the same sample of projects, the COCOMO effort/cost scale factors (prec, flex, resl, team, pmat) have the range:

$$-1.56 \leq m_\beta \leq -1.014 \tag{16}$$

Similarly, there are two sets of defect multipliers and scale factors:

1) The *positive defect* features have slopes m_γ^+ and are proportional to estimated defects. These features are flex, data, ruse, cplx, time, stor, and pvol.
2) The *negative defect* features, with slopes m_γ^-, that are inversely proportional to the estimated defects. These features are acap, pcap, pcon, apex, plex, ltex, tool, site, sced, prec, resl, team, pmat, rely, and docu.

COQUALMO divides into three models describing how defects change in requirements, design, and coding. These tunings options have the range:

$$\begin{array}{l} requirements \left\{ \begin{array}{c} 0 \leq m_\gamma^+ \leq 0.112 \\ -0.183 \leq m_\gamma^- \leq -0.035 \end{array} \right. \\[1em] design \left\{ \begin{array}{c} 0 \leq m_\gamma^+ \leq 0.14 \\ -0.208 \leq m_\gamma^- \leq -0.048 \end{array} \right. \\[1em] coding \left\{ \begin{array}{c} 0 \leq m_\gamma^+ \leq 0.14 \\ -0.19 \leq m_\gamma^- \leq -0.053 \end{array} \right. \end{array} \tag{17}$$

The tuning options for the defect removal features are:

$$\begin{array}{rl} \forall x \in \{1..6\} & SF_i = m_\delta(x-1) \\ requirements: & 0.08 \leq m_\delta \leq 0.14 \\ design: & 0.1 \leq m_\delta \leq 0.156 \\ coding: & 0.11 \leq m_\delta \leq 0.176 \end{array} \tag{18}$$

where m_δ denotes the effect of i on defect removal.