Model-Based Tests of Truisms

Tim Menzies
Lane Dept. of Com. Sci.
University of West Virginia
PO Box 6109, Morgantown
WV, 26506-6109, USA;
tim@menzies.com

David Raffo, Siri-on Setamanit
Portland State University
School of Business
PO Box 751
Portland OR 97207 USA
david@sba.pdx.edu
ssetaman@yahoo.com

Ying Hu, Sina Tootoonian
Dept Elec. & Comp. Eng.
Vancouver, British Columbia
Canada V6T1Z4
huying_ca@yahoo.com
achillesofpersis@hotmail.com

Abstract

Software engineering (SE) truisms capture broadly-applicable principles of software construction. The trouble with truisms is that such general principles may not apply in specific cases. This paper tests the specificity of two SE truisms: (a) increasing software process level is a desirable goal; and (b) it is best to remove errors during the early parts of a software lifecycle.

Our tests are based on two well-established SE models: (1) Boehm et.al.'s COCOMO II cost estimation model; and (2) Raffo's discrete event software process model of a software project life cycle. After extensive simulations of these models, the TAR2 treatment learner was applied to find the model parameters that most improved the potential performance of the real-world systems being modelled.

The case studies presented here showed that these truisms are clearly sub-optimal for certain projects since other factors proved to be far more critical. Hence, we advise against truism-based process improvement. This paper offers a general alternative framework for model-based *assessment of methods to improve software quality:* modelling + validation + simulation + sensitivity. *That is, after recording what is known in a model, that model should be validated, explored using simulations, then summarized to find the key factors that most improve model behavior.*

1 Introduction

What is the best method for improving software quality? Software engineering (SE) research has generated a bewildering range of tools for improving quality. The list of techniques is very, very long and includes model checking [9], static code analysis [21], runtime verification [20], lightweight languages for requirements engineering [36], model-based methods for optimizing software processes [24], just to mention a few.

How can an industrial practitioner hope to choose the right tool from the bewildering set of available alternatives? The SE literature contains many *truisms* that an industrial practitioner might use to select the right tool. For example,

Truism 1: It is best to remove errors during the early parts of a software lifecycle. [36].

Truism 2: Increasing an organization's software process level is a desirable goal [3].

If our practitioner believes in the first truism, then they might investigate Menzies' lightweight requirements engineering languages [36]. Alternatively, if they believe in the second truism, they might reject all automatic software engineering methods and just explore software process improvement techniques such as CMMi [3].

The problem with truisms is that they may be misleading for particular projects, or just plain wrong. The literature is full of evidence that challenges many SE truisms. For example, a dubious truism of visual programming is that "visual representations are inherently superior to mere textual representations". A review by Menzies suggests that the available evidence for this claim is hardly conclusive [30]. Other commonly cited truisms are just as dubious. For example, despite claims that object-oriented (OO) encapsulation will reduce error rates in software [37], empirical results suggest that debugging an OO program is many times harder and longer than debugging a standard procedural program [19]. In other work, Fenton & Neil [13, 14] offer a scathing critique of the truism that "pre-release fault rates for software are a predictor for post-release failures" (as claimed by [10], amongst others). For the systems described in [15], they

[0]ASE, 2002, http://ase.cs.ucl.ac.uk/cfp.html

show that software modules that were highly fault-prone prior to release revealed very few faults after release.

Given this history of failed truisms, do we have any guidance to offer our practitioners trying to select which tool to apply? One method is to study the inherent properties of the tools. For example, Lowry et.al. [27] and Menzies & Cukic [32] contrast the costs and defect detection decay rates of formal methods, white box testing, and black box testing. While such an analysis is useful, our belief is that tool selection must be made on a project-specific basis. This belief results from the experiments described in this paper. Using model-based simulations of software processes, we have identified the changes that most improve the quality of particular software projects. These changes contradict **Truism One** and **Truism Two** (shown above).

It is neither novel nor interesting to say that particular cases can contradict general principles. However, two aspects of our work are both very novel and very interesting. Firstly, our counter-examples to **Truisms One** and **Truisms Two** were found in the *very first case studies* we explored. The ease with which we found these counter-examples makes us very suspicious of these truisms. Secondly, when the truisms generated non-optimum advice, we found we could identify a *minimal* set of changes that *most* improved software projects. Further, using our method, these minimal recommendations are highly specialized to the specific software project being studied. Our method is based on the TAR2 *treatment learner*, described below.

Given unlimited resources, a software project manager may elect to implement TAR2's minimal recommendations *as well as* trying (e.g.) earlier detect removal or increasing their software process. However, in the more usual case of resource-bound software development, software project managers may find it useful to restrict themselves to just TAR2's recommendations since these are the *minimum* changes that *most* improve their projects.

The rest of this paper is structured as follows. Section Two describes our general methods for testing truisms and finding alternate key factors if the truisms fail. That approach can be summarized as follows:

$$decisions = modelling + validation + simulations + sensitivity$$

That is, after recording what is known in a model, that model should be validated, explored using simulations, then summarized to find the key factors that most improve model behavior. Sections Three and Four apply this method to **Truism One** and **Truism Two**. Methods for improving software projects were found that were demonstrably better than the advice offered by the truisms. Finally, in Section Five, we discuss how this kind of analysis might be applied to assessing the relative merits of (e.g.) runtime verification vs model checking.

Before beginning, it is important to stress that it would be a mistake to view our results as new truisms that replace existing truisms. For example, Section Three is *not* an argument that there is *never* any value in software process improvement. Similarly, Section Four is *not* an argument that there is *never* any value in early lifecycle error removal. Early lifecycle defect removal and increasing software process levels can have significant impact on a project. However, what our results are saying is that, for the particular projects studied here, other factors were *more* significant. Further, these more significant factors could be found very simply, using treatment learning.

2 Decisions Using Model-Based Simulations

This section discusses general principles for $modelling + validation + simulations + sensitivity$.

Many general tools and methodologies exist for *modelling* such as distributed agent-based simulations [8], discrete-event simulation [18, 25, 26][1], continuous simulation (also called system dynamics) [1, 45], state-based simulation (which includes petri net and data flow approaches) [4, 17, 29], logic-based and qualitative-based methods [5, chapter 20] [23], and rule-based simulations [38]. Note that for software process programming, elaborate new modelling paradigms may not be required. For example, the Little-JIL process programming language [6, 46] just uses standard programming constructs such as pre-conditions, post-conditions, exception handlers, and a top-down decomposition tree showing sub-tasks inside tasks.

Simulations can be based on *nominal* or *off-nominal values*. Nominal simulations draw their inputs from known operational profiles of system inputs [39]. Off-nominal monte-carlo (also called stochastic) simulations, where inputs are selected at random, can check for unanticipated situations [16]. Stochastic simulation has been extensively applied to models of software process [42]. It is strange to report, but examples of execution of Little-JIL are rare (an interpreter exists for Little-JIL models but examples of its execution are not mentioned (i.e. in [6,46]).

In a *sensitivity* analysis, the key factors that most influence a model are isolated. Also, recommended settings for those key factors are generated. We take care to distinguish sensitivity analysis from traditional optimization methods. In our experience, the real systems we deal with are so complex that they do not always fit into (e.g.) a linear optimization framework. Studying data grown from simulators lets us investigate complex, non-linear systems using a variety of data driven distributions. These models can capture complex feedback and rework loops which are not possible for traditional optimization methods. Our experience is that simulation models can look at processes in detail as well as

[1] See also the `http://www.imaginethatinc.com` web site.

at a high level of abstraction which is where the more analytic models must reside. Finally, simulation models can capture multiple performance measures not able to be explored using these optimization formulations. This is not to say that traditional optimization models are not useful. For certain questions, traditional optimization formulations provide the best fit for the question that is trying to be answered. However, for many questions, the standard optimization models are not the best choice and something like *treatment learning* may be more useful.

2.1 Treatment Learning

Treatment learning is our preferred general method for multi-dimensional sensitivity analysis [12, 22, 31, 33–35]. Treatment learners find the *fewest* factors that *most* influence a simulation model. In the case of simulations generated from models of software process, these summaries can read as advice on how to best optimize a software process.

The TAR2 treatment learning has been successfully applied either in a single batch mode [22, 33–35], or incrementally in order to encourage a system towards some desired goal [12, 31]. The algorithm makes no assumption about continuity of variables and hence may succeed where standard regression and linear optimization may fail. Conceptually, given a set of input attributes and output classifications, the algorithm searches through all combinations of attribute ranges that are under consideration to find those which lead to the *most* desired outputs and the *least* undesirable outputs. This search is clearly intractable since a complete search of all subsets of the input attribute ranges would take exponential time. TAR2 only works since a linear-time heuristic scoring mechanism can quickly find which attribute ranges can be ignored.

As explained in [35], the algorithm is a *minimal contrast set association rule learner with weighted classes*. That is, while standard machine learners find possibly complex descriptions of the classes within a domain, TAR2 finds the minimal difference *between* classes (the contrast set). The algorithm uses a heuristic class weight in order to report the *least* change that selects for the *most* preferred classes.

The standard introductory example for TAR2 (e.g. as shown in [31]) is the log of golf playing behavior of Figure 1. This log contains four attributes and 3 classes. TAR2 accesses a *score* for each class. For a golfer, the classes in Figure 1 could be scored as *none=2* (i.e. worst), *some=4*, *lots=8* (i.e. best). TAR2 seeks *outstanding* attribute ranges; i.e. those that occur far more frequently in the highly scored classes than in the lower scored classes.

Treatments are formed from subsets of the outstanding attribute ranges and applied to the example data. Examples that contradict the proposed treatment are rejected. The *worth* of a treatment is assessed by comparing the *baseline* class distribution to the *treated* distribution. For example, Figure 2 shows TAR2's analysis of the golf data. The left-hand-side histogram of Figure 2 shows the baseline class frequency in Figure 1. Note that in the baseline, we only play golf lots of times in $\frac{6}{5+3+6} = 43\%$ of cases. The middle histogram of Figure 2 shows the best action found by TAR2: with the restriction that *outlook=overcast*, then we play golf lots of times in 100% of cases. The right-hand-side histogram of Figure 2 shows the worst action found by TAR2. Such a worst-case scenario can be generated by reversing the class scores. This reversal makes TAR2 seek the *worst* possible treatment. In the case of our golf example, with the restriction that $humidity \geq 90$ then we play lots of golf in $\frac{1}{3+1+1} = 20\%$ of cases. In summary, to maximize golf playing behavior, we should select a holiday location with an overcast outlook. While on holidays, we need only monitor the humidity (not temperature or outlook or wind) and become alarmed if the humidity increases over 90%.

TAR2's treatments should be assessed on examples not seen during training. There are two standard methods for doing so: *n-way cross validation* and *re-simulation*. In the

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

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

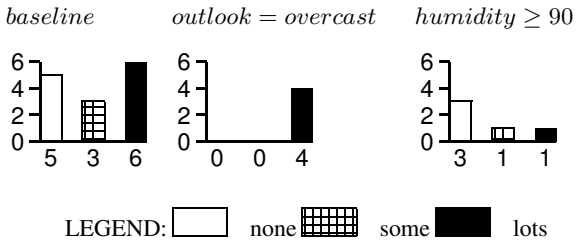

Figure 2. Treatments that can change golf playing behavior from the baseline.

former, the training set is divided into N buckets. For each bucket in turn, a treatment is learned on the other $N-1$ buckets then tested on the bucket put aside. A treatment is deemed *stable* if it works in the majority of all N turns. The results of our first case study will be assessed via N-way cross validation. In a re-simulation study, the treatments recommended by TAR2 are imposed on the simulator. The simulator is then run again. A treatment is deemed *predictive* if the predicted distribution is realized in output of a simulator constrained to TAR2's proposed treatment. The results of our second case study will be assessed via re-simulation.

3 Case Study 1: "Increasing an organizations software process level is a desirable goal"

In this section, we apply our $modelling + validation + simulation + sensitivity$ approach to the truism: "increasing software process level is desirable".

The Software Engineering Institute's (SEI) capability maturity model (CMM [40]) categorizes software organizations into one of five process levels. Below CMM5, there is no use of measurements to optimize a company's software process. Also, below CMM4, there is no systematic data collection. Lastly, below CMM3, a company's software process is not even written down.

The CMM5 movement has many leaders (e.g. [3]) but few followers. In 1996, less than 12% of one sample of certain software organizations were above level 3[2]. Further, the current industry average seems to be less than CMM2[3].

In order to assess the merits of increasing software process, we applied the formula $modelling + validation + simulation + sensitivity$ to one NASA project. Eleven changes to that project were considered, including an increase to the software process level. Counter to the truism of this section, TAR2 found that the best action included a *low* software process level. The rest of this section describes that study.

3.1 Modelling

Figure 3 shows a NASA software project scored on the 22 parameters of the COCOMO-II software cost estimation model [2][4]. The core intuition of COCOMO-II is that the effort required to develop software increases exponentially as that software grows in size; i.e.:

[2] `http://www.telcordia.com/newsroom/mediaclips/telossource/telesrcanalyst.html`

[3] Personal communication with SEI researchers.

[4] For a precise definition of these parameters, see `http://sunset.usc.edu/research/COCOMOII/expert_cocomo/drivers.html`

	ranges		KC-1 now	KC-1 changes
Scale drivers	prec = 0..5	precedentness	0, 1	
	flex = 0..5	development flexibility	1, 2, 3, 4	1
	resl = 0..5	architectural analysis or risk resolution	0, 1, 2	2
	team = 0..5	team cohesion	1, 2	2
	pmat = 0..5	process maturity	0, 1, 2, 3	3
Product attributes	rely = 0..4	required reliability	4	
	data = 1..4	database size	2	
	cplx = 0..5	product complexity	4, 5	
	ruse = 1..5	level of reuse	1, 2, 3	3
	docu = 0..4	documentation requirements	1, 2, 3	3
Platform attributes	time = 2..5	execution time constraints	?	
	stor = 2..5	main memory storage	2, 3, 4	2
	pvol = 1..4	platform volatility	1	
Personnel attributes	acap = 0..4	analyst capability	1, 2	2
	pcap = 0..4	programmer capability	2	
	pcon = 0..4	programmer continuity	1, 2	2
	aexp = 0..4	analyst experience	1, 2	
	pexp = 0..4	platform experience	2	
Project attributes	ltex = 0..4	experience with language and tools	1, 2, 3	3
	tool = 0..4	use of software tools	1, 2	
	site = 0..5	multi-site development	2	
	sced = 0..4	time before delivery	0, 1, 2	2

Figure 3. COCOMO-II parameters. Scale drivers are listed first. The cost drivers are union of the product, platform, personnel, and project attributes. Last two columns show values known within one NASA software project.

$$a * \left(KSLOC^{\left(1.01+\sum_{i=1}^{5} SF_i\right)} \right) * \left(\prod_{j=1}^{17} EM_j \right)$$

This expression computes *person months* for a project; i.e. 152 hours of effort (and includes development and management hours). In this expression, a is a domain-specific parameter; KSLOC is estimated directly or computed from a function point analysis; SF_i are the scale drivers (e.g. drivers such as "have we built this kind of system before?"); and EM_j are the cost drivers (e.g. required level of reliability). Figure 3 lists the COCOMO-II scale drivers and cost drivers. COCOMO-II also defines a similar equation for the recommended number of *person months* for the project. We elected to report the COCOMO-II output expressed as the number of *staff* required to get *person months* of work done in the recommended number of *months*:

$$staff = \frac{person\ months}{months}$$

	rely= very low	rely= low	rely= nominal	rely= high	rely= very high
sced= very low	0	0	0	1	2
sced= low	0	0	0	0	1
sced= nominal	0	0	0	0	0
sced= high	0	0	0	0	0
sced= very high	0	0	0	0	0

Figure 4. A Madachy table. From [28]. This table reads as follows. In the exceptional case of high reliability systems and very tight schedule pressure (i.e. *sced=low or very low* **and** *rely= high or very high*), **add some increments to the built-in parameters (increments shown top-right). Otherwise, in the non-exceptional case, add nothing to the built-in parameters.**

The column labelled *now* in Figure 3 shows the current situation of a particular NASA project. The analysts interviewed for this case study knew some uncertainties existed in their understanding of this project. Where somewhat uncertain, they used ranges; e.g. it was unclear if developers had never seen this kind of application before so $prec = \{0, 1\}$. When totally uncertain, they just used a question mark; e.g. no knowledge about execution time constraints was available so $time = ? = \{2, 3, 4, 5\}$ where $\{2, 3, 4, 5\}$ is the complete range of possible values for $time$.

In this study, inputs such as Figure 3 are processed by the COCOMO-II model as well as the Madachy model of software project management issues [28]. While the COCOMO-II model estimates development effort, the Madachy model outputs a numeric index representing how concerned an experienced analyst might be about a particular software project. The model contains 94 tables that implement a context-dependent modification to internal COCOMO parameters. Figure 4 operationalizes one of the 94 Madachy heuristics: i.e. software that must be highly reliable should not be developed under excessive schedule pressure. While Madachy calls his work a "risk" model, his definition is so different to the standard definition of $risk = severity * frequency$ that we rename it to a "worries" model.

COCOMO-II assesses process maturity, or $pmat$, via an 18-point questionnaire that explores the key process areas of a project. Since most projects score very low on the CMM scale, COCOMO-II divides CMM1 into two zones. Hence, the COCOMO-II's $pmat$ range is $\{0,1,2,3,4,5\}$ where 0,1 denote the lower and upper half of CMM1 (respectively). Observe the values for process maturity in Figure 3. Our analysts said that $pmat = \{0, 1, 2, 3\}$; i.e. this particular project's process level was less than CMM3.

The column labelled *changes* in Figure 3 shows eleven proposed changes to the current situation. Note that it is proposed to change process level up to COCOMO-II $pmat = 3$. The goal of our sensitivity study will be to assess if $pmat = 3$ is the preferred process level.

3.2 Validation

Before trusting decisions from $modelling + validation + simulation + sensitivity$, it is important to validate the model. The COCOMO team has published numerous validation studies of COCOMO models. For example, in ten trials with data not used during model tuning, the COCOMO-II effort prediction came within 25% of the actual effort in 69% (on average) of cases (this is denoted as $PRED(25) = 69$) [7]. In other validation work from the COCOMO team, Madachy has reported studies [28] with his model and the COCOMO-I project database. Those studies showed a good correlation between the Madachy "worries" index and $\frac{months}{KDSI}$ (where KDSI is thousands of delivered source lines of code). That is, the Madachy model's output is a measure of the danger that a project will take longer than planned to build.

3.3 Simulation

COCOMO models require an internal table of numeric values that map (e.g.) $pmat = 2$ into the effort and schedule equations. This study used the latest published COCOMO-II table as shown in [7].

Model inputs were selected at random, 10,000 times, from the parameter ranges shown in Figure 3's *now* column. In this study, some uncertainty existed in the size estimates and so the SLOC (source lines of code) estimate was taken to be 75K, 100K, 125K. Hence, the model was run 30,000 times (10,000 times for each SLOC value).

Model outputs for *staff* and *worries* were computed using COCOMO-II and the Madachy model (respectively). Figure 5 shows the results as a *percentile matrix*; i.e. it shows what percentage of the 30,000 runs falls into a particular range. The percentiles matrix is color-coded: the darker the cell, the large the percentage of the runs in that cell. As might be expected from the large range of possible input values shown in Figure 3, there is a large variance in the results.

3.4 Sensitivity

TAR2 was used to find which subset of the proposed changes had the most impact on the project; i.e. most decreased staff levels and the "worries" index. Treatments that did not include ranges outside of the *changes* column of

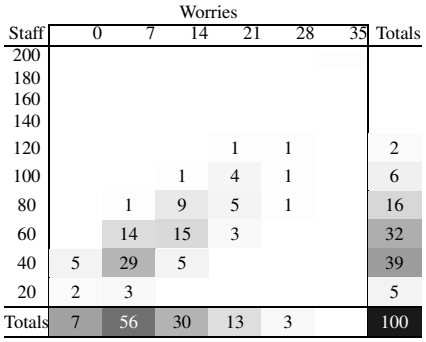

Figure 5. Simulation outputs using inputs specified in Figure 3. Each cell shows the percentage of the runs that fall into a certain range.

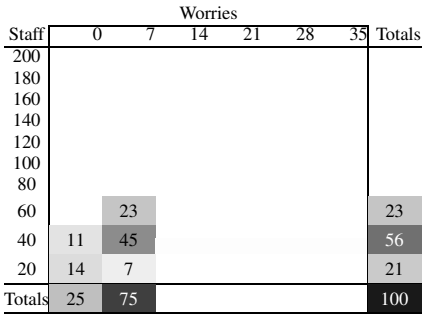

Figure 6. Sensitivity results for Figure 5.

Figure 3 were rejected. After exploring all subsets of the proposed changes, TAR2 found the following treatment:

$$pmat = 1 \wedge acap = 2 \wedge sced = 2$$

Further experimentation showed that no other treatment had a better impact. That is, the best treatment is this case was a combination of (i) increasing the time to delivery to 100% of the time proposed by the project- i.e. no pressure for an early delivery; (ii) using analysts with a middle-range of ability (fall between the 45th to 65th percentile); and (iii) ensuring that the project was at least in the upper-half of CMM1, but don't go to CMM process level 2.

This treatment was tested via *re-simulation*. The input ranges for *pmat,acap,sced* were set as above, and the rest of the inputs were left as before; i.e. able to range over all the values of Figure 3. The results are shown in Figure 6. When compared to Figure 5, it can be seen that the proposed treatments greatly *reduced* the variance in the model's behavior while *improving* the mean values (decreased staffing level and"worries").

Note that, contrary to the truism that process level improvement is an invaluable method of improving software development, this particular project needed only a very minimal level of CMM process (upper half of CMM1).

4 Case Study 2: "It is best to remove errors during the early software lifecycle"

In this section, we apply our $modelling + validation + simulation + sensitivity$ approach to the truism: "It is best to focus on error removal during early lifecycle".

Many researchers have predicated their work on the truism that catching errors during early software lifecycle is very important. For example, the first author has written [36]:

> The case for more formality in (early lifecycle) is overwhelming. Many errors in software can be traced back to errors in the requirements [43]. Often, the conception of a system is improved as a direct result of the discovery of inadequacies in the current conception. The earlier such inadequacies are found, the better, since the cost of removing errors at the earlier stage can be *orders of magnitude cheaper* than the cost of removing errors in the final system [44].

However, as we shall see in the following study, removing early lifecycle errors for a particular project was not nearly as important as selection of the inspection methods

4.1 Modelling

In this section we will describe a particular simulation model tuned to a particular company, as well as the project and development context of that model. A high-level block diagram of this model is shown in Figure 7. The model is far more complex that suggested by Figure 7 since each block references many variables shared by all other blocks.

This model was originally developed in 1995 [41] and was subsequently tailored to a specific large-scale development project at a leading software development firm with the following properties:

- A large company with a number of development sites throughout the world.
- Between 10 and 20 major projects being conducted at one time at the study site.
- Work scope included world-wide system support of most of the products being developed at the site (including the one being studied). Support professionals are actually experienced developers who have been assigned to correct field detected defects. When these support professionals have time available, they carry out limited development tasks.

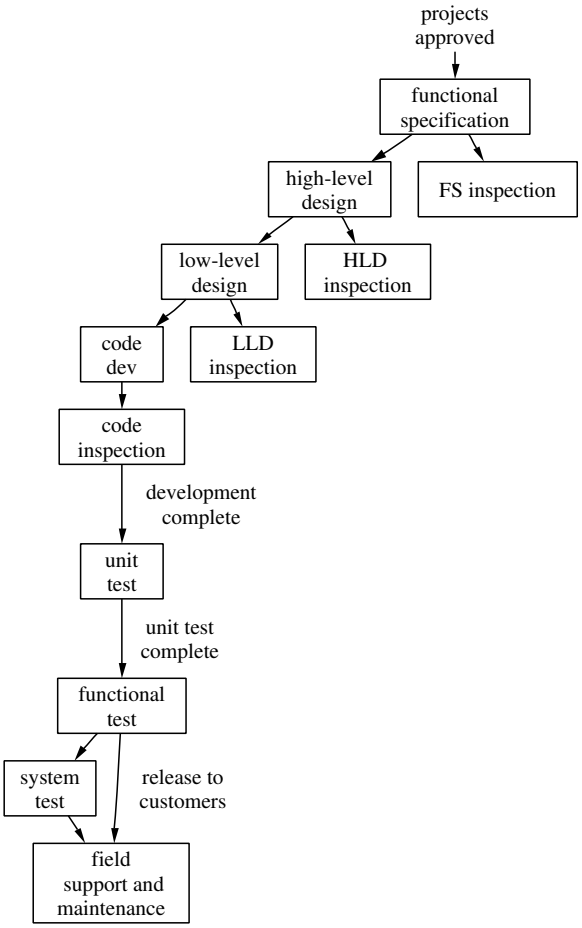

Figure 7. High-level block diagram of a discrete event model of one company's software process.

- The site achieved ISO certification and was assessed at a Level 2 using a CMM-SPA during the study period.
- The product studied had completed 5 successive major releases when the study began.
- In each release, major new functionality was added and substantial revisions of existing functionality occurred.
- At peak development periods the project involved over 70 people.

The software process studied at this company essentially followed a waterfall process model including the major phases of functional specification (FS), high-level design (HLD), low-level design (LLD), coding (CODE), unit test (UT), functional verification test (FVT), and system verification test (SVT). Inspections of the functional specification, high-level design, low-level design, and code were also conducted. After SVT and FVT were completed the product was released to the public. These phases, as well as a period devoted to field support and maintenance are captured by this model. In addition, the process segments for developing test plans and writing test cases for functional test and system test were also included.

We developed two models of this process - a state-based simulation model using the Statemate Magnum tool by i-Logix [5] and a discrete event model using the Extend simulation language [6]. The discrete event model contained 30+ process steps with two levels of hierarchy. The main performance measures of development cost, product quality and project schedule were computed by the model. These performance measures could also be recorded for any individual process step as desired. 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 / effect of training provided; and resource constraints.

Actual data were used for model parameters where possible. For example, inspection data was collected from individual inspection forms for the two past releases of the product. Distributions for defect detection rates and inspection effectiveness where developed from these individual inspection reports. Also, effort and schedule data were collected from the corporate project management tracking system. Lastly, senior developers and project managers were surveyed and interviewed to obtain values for other project parameters when hard data were not available.

Models were developed from this data using multiple regression to predict defect rates and task effort. The result was a model that predicted the three main performance measures of cost, quality, and schedule. A list of all of the process modification supported by the model are too numerous to list here. Suffice to say that small to medium scope process changes could be easily incorporated and tested using the model.

4.1.1 Inspection Types

For each phase of the modelled process, there was a choice of either having one of four inspection types; e.g. none, or a *full fagan*, or a *baseline* inspection, or a *walk through*. These terms are explained below.

A *full fagan inspection* [11] is a well-researched manual inspection method. Such inspections are precisely defined, including a seven step process plus pre-determined roles for inspection participants. Fagan inspections can find many

[5] Statemate and I-Logix are registered trademarks ® of I-Logix Inc. (3 Riverside Drive; Andover, Massachusetts 01810 USA).

[6] Extend and Imagine That are registered trademarks ® of Imagine That, Inc. (6830 Via Del Oro, Suite 230, San Jose, California, 95119 USA).

errors in a software product. For example, for the company studied here, the *defect detection capability*[7] of their full Fagan inspections was $TR(0.35, 0.50, 0.65)$[8]. In this study, a full Fagan inspection used between 4 and 6 staff, plus the author of the artifact being inspected.

A *baseline inspection* was a continuation of current practice at the company under study. The baseline inspection at this company was essentially a poorly performed Fagan inspection, The distinction between a proper Fagan inspection and the baseline is that staff would receive new training, checklists and support in order to significantly improve the effectiveness of the inspections. The data showed that baseline inspections had varying defect detection capabilities ranging from a minimum of 0.13, a maximum of 0.30 and an average of 0.21 (these figures were obtained from actual inspection records).

Walk through inspections were conducted by one by the author of the artifact being inspected in a relatively informal atmosphere. Process experts estimated the amount of time and defect detection capability for this type of inspection. Those estimates were $TR(0.07, 0.15, 0.23)$.

4.1.2 Summarizing Model Output

The outputs of the model are assessed via a multi-attribute utility function

$$\begin{aligned} utility &= 40 * (14 - quality) + \\ &\quad 320 * (70 - expense) + \\ &\quad 640 * (24 - duration) \end{aligned} \quad (1)$$

where *quality, expense* and *duration* are defined as follows. *Quality* is the number of major defects (i.e. severity 1 and 2) estimated to remain in the product when released to customers. *Expense* is the number of person-months of effort used to perform the work on the project and to implement the changes to the process that were studied. *Duration* is the number of calendar months for the project from the beginning of functional specification until the product was released to customers.

The justification for this style of utility function is discussed in detail in [41]. In summary, this function was created after extensive debriefing of the business users who funded the development of this model.

4.2 Validation

The baseline simulation model of the lifecycle development process seen in Figure 7 was validated in a number of ways. The most important of which were as follows. In *face validity* studies, 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 for their fidelity to the actual. In *output validity* 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. This result was not accepted initially by management as it was too long, however, upon further investigation it was found that the model predictions corresponded quite accurately with this company's actual experience.

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

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

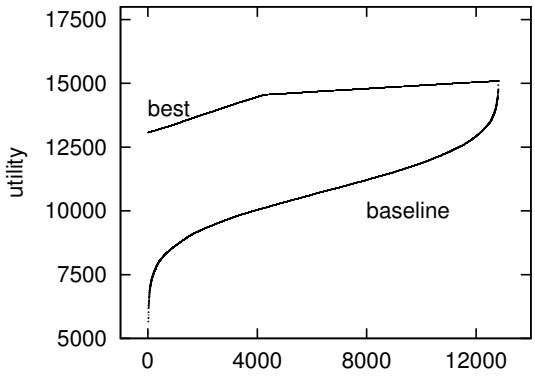

Figure 8. Sorted utilities generated in case study 2.

4.3 Simulation

The simulation model described above contains four phases of development and four inspection types at each phase. The four phases were functional specification (FS); high level design (HLD); low level design (LLD); and coding (CODE). The four inspection types were full Fagan, baseline, walk through, and none. This results in 4^4 different configurations. Each configuration was executed 50 times, resulting in $50 * 4^4 = 12800$ runs. Each run was summarized using Equation 1. The utilities generated by this method are shown sorted as the *baseline* plot of Figure 8. Note the huge range of output values: 5,000 to 15,000.

4.4 Sensitivity

TAR2 was used to explore the simulation data. The best treatment found was to use the following configuration:

- No functional specifications inspections;
- Baseline inspections for high level design; i.e. no change from current practice;
- Full Fagan for low level design and code reviews.

Further experimentation showed that no larger treatment had a greater impact. That is, other factors in the Figure 7 model were not as influential as the selection of inspection type. Given this configuration, TAR2 predicted the distribution of utility values seen as the *best* plot of Figure 8. When compared to the *baseline* plot, we see that this treatment results in a significant improvement in model behavior. Also, the variance in the utilities has been greatly reduced.

This treatment was assessed via *10-way cross validation*. The *best* plot of Figure 8 was observed to be the average improvement seen under cross validation.

Note that contrary to the truism that early lifecycle detection and error removal is valuable, this particular project was most impacted by spending *less* effort on early lifecycle inspections and *more* effort on late lifecycle inspections.

5 Discussion

Opponents of our approach might argue that it is not as simple as it appears. For example, our approach needs a model of the software project and building such models can be a non-trivial task. There are two replies to this objection. Firstly, our approach does not always require elaborate modelling. For example, the model used in Section Three was an off-the-shelf open-source model; i.e. it did not require any effort to develop. Hence, our recommendation is that if resources permit, then detailed models should be built. Otherwise, public domain models can suffice- at least for an initial study. Secondly, this work shows that truism-based changes to software processes may not achieve the benefits they promise. The extra modelling cost required by our method must be compared to the cost of designing and implementing useless process changes that do not deliver the promised benefits.

Our paradigm of $decisions = modelling + validation + simulations + sensitivity$ gives us the ability to examine the conditions under which the performance of a given system may be dramatically improved. Moreover, this approach provides prescriptive guidance by identifying the ranges to which various input parameters (such as inspection efficiency and the like) much be moved in order to achieve these desired levels of performance. Ideally, we would like to extend the current study to assessing standard automated software engineering methods.

For example, in the following scenario we would be able to assess methods such as runtime verification, model checking, or static analysis:

- In a particular project, it is determined that a particular range of defect detection in source code is essential for most improving overall software quality.
- It is determined that range is too high for manual methods such as Fagan inspections.
- Using historical data, it is possible to select the automated software engineering method(s) that achieves the target defect detection range.
- The selected automated software engineering methods are assessed by checking which is cheapest to deploy in the current project.

Note that this study would require information on the detect detection capability of the different automatic software engineering tools. To the best of our knowledge, such information is unavailable. Hence, at this time, the above scenario can't yet be performed.

References

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

[2] C. Abts, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece. COCOMO II model definition manual. Technical report, Center for Software Engineering, USC,, 1998. `http://sunset.usc.edu/COCOMOII/cocomox.html#downloads`.

[3] D. Ahern, A. Clouse, and R. Turner. *CMMI Distilled*. Addison-Wesley, 2001.

[4] 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.

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

[6] 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)*, pages 754–757, June 2000.

[7] S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transaction on Software Engineerining*, 25(4), July/August 1999.

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

[9] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] H. Dunsmore. Evidence supports some truisms, belies others. (some empirical results concerning software development). *IEEE Software*, pages 96–99, May 1988.

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

[12] 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://tim.menzies.com/pdf/02re02.pdf.

[13] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. Available from http://citeseer.nj.nec.com/fenton99critique.html.

[14] N. E. Fenton and M. Neil. Software metrics: A roadmap. In A. Finkelstein, editor, *Software metrics: A roadmap*. ACM Press, New York, 2000. Available from http://citeseer.nj.nec.com/fenton00software.html.

[15] N. E. Fenton and S. Pfleeger. *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[16] W. Gutjhar. Partition vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engineering*, 25(5):661–674, September/October 1999.

[17] D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.

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

[19] L. Hatton. Does oo sync with how we think? *IEEE Software*, pages 46–54, May/June 1998.

[20] K. Havelund and T. Pressburger. Model checking java programs using java pathfinder. *International Journal on Software Tools for Technology Transfer*, 2(4), April 2000. Available from http://ase.arc.nasa.gov/visser/jpf/jpf1.ps.gz.

[21] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 1(12):26–60, January 1990.

[22] Y. Hu. Better treatment learning, 2003. Masters Thesis, Department of Electrical Engineering, University of British Columbia, in preperation.

[23] Y. Iwasaki. Qualitative physics. In P. C. A. Barr and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, pages 323–413. Addison Wesley, 1989.

[24] M. Kellner, R. Madachy, and D. Raffo. Software process modeling and simulation: Why, what, how,. *Journal of Systems and Software*, 46(2/3), 1999.

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

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

[27] M. Lowry, M. Boyd, and D. Kulkarni. Towards a theory for integration of mathematical verification and empirical testing. In *Proceedings, ASE'98: Automated Software Engineering*, pages 322–331, 1998.

[28] R. Madachy. Heuristic risk assessment using cost factors. *IEEE Software*, 14(3):51–59, May 1997.

[29] 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.

[30] T. Menzies. Evaluation issues for visual programming languages. In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*. World-Scientific, 2002. Available from http://tim.menzies.com/pdf/00vp.pdf.

[31] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In *Annals of Software Engineering (submitted)*, 2002. Available from http://tim.menzies.com/pdf/02itar2.pdf.

[32] T. Menzies and B. Cukic. How many tests are enough? In S. Chang, editor, *Handbook of Software Engineering and Knowledge Engineering, Volume II*, 2002. Available from http://tim.menzies.com/pdf/00ntests.pdf.

[33] T. Menzies and Y. Hu. Reusing models for requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from http://tim.menzies.com/pdf/01reusere.pdf.

[34] T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, *Formal Approaches to Agent-Based Systems, book chapter*, 2002. Available from http://tim.menzies.com/pdf/01agents.pdf.

[35] T. Menzies and Y. Hu. Just enough learning (of association rules). In *WVU CSEE tech report*, 2002. Available from http://tim.menzies.com/pdf/02tar2.pdf.

[36] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via 'topoi diagrams'. In *ICSE 2001*, 2001. Available from http://tim.menzies.com/pdf/00fastre.pdf.

[37] B. Meyer. *Object-oriented Software Construction*. Prentice-Hall,Hemel Hemstead, 1988.

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

[39] J. Musa. *Software Reliability Engineered Testing*. McGraw-Hill, 1998.

[40] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model, version 1.1. *IEEE Software*, 10(4):18–27, July 1993.

[41] D. Raffo. Modeling software processes quantitatively and assessing the impact of potential process changes of process performance, 1995. Ph.D. thesis, Manufacturing and Operations Systems, Carnegie Mellon University.

[42] D. M. Raffo, J. V. Vandeville, and R. Martin. Software process simulation to achieve higher cmm levels. *Journal of Systems and Software*, 46(2/3), April 1999.

[43] D. Reifer. Software failure modes and effects analysis. *IEEE Transactions on Reliability*, pages 247–249, 1979.

[44] F. Schneider, S. Easterbrook, J. Callahan, G. Holzmann, W. Reinholtz, A. Ko, and M. Shahabuddin. Validating requirements for fault tolerant systems using model checking. In *3rd IEEE International Conference On Requirements Engineering*, 1998.

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

[46] 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.