Better reasoning about software engineering activities

Tim Menzies[1]
University of British Comlumbia
tim@menzies.com

James D. Kiper[2]
Miami University
kiperjd@muohio.edu

ABSTRACT

Software management oracles often contain numerous subjective features. At each subjective point, a range of behaviors is possible. *Stochastic simulation* samples a subset of the possible behaviors. After many such stochastic simulations, the TAR2 *treatment learner* can find control actions that have (usually) the same impact despite the subjectivity of the oracle.

1. Introduction

Experience with software management can be encoded into an management *oracle* that can offer advise on how to structure a software project. Three components are required for such an oracle. Firstly, a *knowledge source* is required. Secondly, a *syntax* is required to encode that knowledge source. Thirdly, an interpreter is required to execute the syntax. In the case of subjective software engineering knowledge, this interpreter must be able to handle *degrees of belief*.

This paper reports experience with one such oracle. The *knowledge source* was a detailed description of CMM2 [3, p125-191]. We used CMM2 since, in our experience, many organizations can achieve at least this level. CMM2 is less concerned with issues of (e.g.) which design pattern to apply, than with what overall project structure should be implemented. Improving CMM2-style decisions is important since in early software lifecycle, many CMM2-style decisions effect the resource allocation for the rest of the project.

The JANE language was used to *encode* the knowledge source. In order to handle *degrees of belief*, JANE assigns a `Chances` weight to all of its propositions. This weight is a subjective judgement and so, for each run, the JANE interpreter hence varies this weight according to distributions supplied by the analyst. After many such runs, a large log of possible behaviors is generated which is summarized by the TAR2 *treatment learner*.

Despite the range of possible actions within our encoding of CMM-2, and the subjectivity of the weights measures, there exists a small number of actions that have a significant impact on the project. Figure 1 shows three sets of actions learnt by TAR2. The left-hand-side histogram marked Δ_0 shows the ratio of different project types predicted by our software management oracle. The other histograms ($\Delta_{i>0}$) show how those ratios change after applying the *treatments* learnt by TAR2; i.e. managers taking certain actions to change their current situation. The *worth* of each option is a reflection of the proportion of good and bad projects, compared to Δ_0; i.e. ($worth(\Delta_0) = 1$). Note that as *worth* increases, the proportion of preferred projects also increases.

The rest of this paper describes how Figure 1 was generated.

2. JANE

A JANE programmer enters in models in a propositional rule-based language. Internally, these rules are converted into a directed graph with the following BNF[1]:

```
Graph  ::== Goal (Vertex)* (Edge)*
Goal   ::== Item
Vertex ::== Mix | Item
Item   ::== Type  Variable Value Label
Type   ::== action | fault | requirement
Mix    ::== CombineRules  Order
```

Note that each `Graph` has a special `Goal Item`. Conceptually, JANE is a backward chainer that performs a recursive descent from the `Goal`.

In JANE, every vertex is either an `Item` where some `Value` is assigned to some `Variable`; or a `Mix` vertex

[1]Proceedings of ASE 2001, San Diego, USA. Corresponding author. Department of Electrical & Computer Engineering; 2356 Main Mall; Vancouver, B.C. Canada V6T1Z4. Phone: (604) 822-3381 Web: `http://tim.menzies.com`.

[2]Dept. Computer Science & Systems Analysis, Miami University, Oxford, Ohio, USA.

[1]In this article's BNF notation, `W ::== X Y | Z` denotes that the structure W contains either the structures `X` and `Y` or the structure Z. `(X)*` denotes zero or more repeats of `X`. `(X)+` denotes one or more repeats of `X`. `[X]` denotes that `X` is optional. Terminals start with lower case, or are quoted

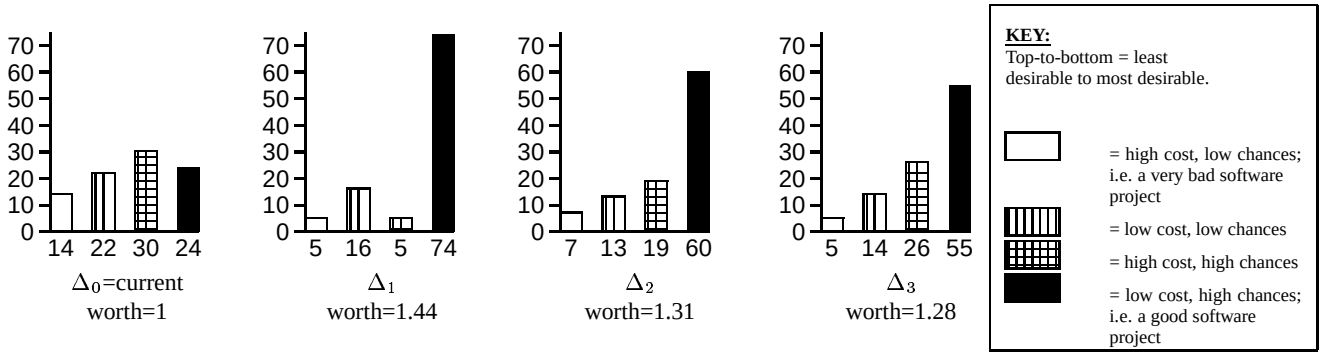

Figure 1. Ratios of different software project types seen in four situations.

where influences are combined. Each `Vertex` and `Edge` in JANE is augmented with a `Cost` and a `Chances` weight and `Mix` vertices define how `Cost`s and `Chances` are combined"

```
Edge    ::== Vertex Label Vertex
Label   ::== Cost Chances
Cost    ::== null | 0.00 .. infinity
Chances ::== null | 0.00 .. 1.00
```

JANE's `Chances` define the extent to which a belief in one vertex can propagate to another. `Cost`s let an analyst model the common situation where some of the `Cost` of some procedure is amortized by reusing its results many times. Hence, the *first* time we use an `action` we incur that `Cost` but afterwards, that `action` is free of charge.

The `Cost` and `Chances` of `Vertex`s are either provided by the JANE programmer or computed at runtime via a traversal of the edges. In JANE, this computation is defined by the `CombineRules` at the `Mix` vertices. For each child of a `Mix` node, a recursive descent is executed and the returned `Cost` and `Chances` values are combined according to the `CombineRules`:

```
CombineRules   ::== CostCombine
                 | ChancesCombine
CostCombine    ::== first(cost)
                 | sum(cost)
ChancesCombine ::== first(chances)
                 | sum(chances)
                 | negate
                 | product(chances)
```

`Negate` is used for negation. For example, when searching *X if not A*, the `Chances` of *X* are *1-Chances(A)*. `Product(chances)` is used for *conjunctions*. For example, when searching *X if A* `and` *B* `and` *C*, the `Chances` of *X* is the product of the chances of *A,B,C*. `First(X)` is used for *simple disjunctive evidence*. For example, when testing *X if A* `or` *B* `or` *C*, then the `Cost` and `Chances` of *X* is taken from the first member of *A,B,C* that is satisficed. `Sum(X)` is used for *summing disjunctive evidence*. For example, JANE supports a special operator called `ors` that is used for implementing disjunctive summing (JANE also supports several other novel operators, described in the next section). `Ors` is like `or` except that when testing (e.g.) *X if A* `ors` *B* `ors` *C*, all members of *A,B,C* will tested. If at least one succeeds, the the `Cost` and `Chances` of *X* is summed from the satisficed members of *A,B,C*.

`Sum(X)` and `Product(X)` can be combined to implement risk mitigation effects. For example, suppose some *action5* disables most of the contribution of *fault3* on *fault1*. In the JANE syntax, this can be coded as

```
fault1 if fault2 @ 0.4
    ors (fault3@0.4 and not action5@0.9)
```

The `ors` operator uses `Sum(chances)`; `and` uses `Product(chances)`; and `not` uses `negate`. Hence, if the `Chance`s of *fault2,fault3,action5* are all unity, then of *action5* is false, the `Chance`s of *fault1* is $0.4 + (0.4 * (1 - 0)) = 0.8$. Alternatively, if *action5* is true, the `Chance`s of *fault1* is $0.4 + (0.4 * (1 - 0.9)) = 0.44 \ll 0.8$. That is, using *action5* significantly reduces the `Chance`s of *fault1*.

3. JANE and Random Search

JANE supports two mechanisms for exploring the space of possibilities within the CMM-2 encoding. Both are random search tools. Firstly, when defining `Cost`s and `Chances`, the programmer can supply a *range* and a *skew*. For example:

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

defines the *cost* of *goodUnitTesting* as being somewhere in the range 1 to 5, with the mean skewed slightly towards 5 (denoted by the "+"). During a simulation, the first time this cost is accessed, it is assigned randomly according to the

range and skew. The assignment is cached so that all subsequent accesses use the same randomly generated value. After each simulation, the cache is cleared. After thousands of simulations, JANE can sample the "what-if" behavior resulting from different assignments within the range.

Secondly, JANE can randomized where it searches using the `Order` of the `Mix` operators:

```
Order      ::== Random | Left2Right
Left2Right ::== and  |  or | ors  |  any | not
Random     ::== rand | ror | rors | rany
```

In the case where the child-order represents some sequence that must be preserved (e.g. ordering software processes) JANE programmers can use `Left2Right` operators. However, repeatedly searching a graph in the same order may miss important interactions. In order not to miss such important interactions, JANE supports several random search operators: `rand`, `ror`, `rors`, and `rany`. For $op \in \{rand, ror\}$, JANE tries to prove all of *X op Y op Z*, but does so in some randomly selected order.

`Rany` is best understood by comparison with `ror`. The expression *X* `ror` *Y* `ror` *Z* is exited when any one of *X,Y,Z* is satisficed. In contrast, JANE tries to prove one *or more* of *X* `rany` *Y* `rany` *Z*, and does so in some randomly selected order. `Rany` is useful when searching for subsets that contribute to some conclusion. For example, the following JANE rule offers several essential features of *stableRequirements* plus several optional factors relating to monitoring change in evolving projects. The essential features are `rand`-ed together while the optional factors are `rany`-ed together.

```
stableRequirements   if effectiveReviews @ 0.3
    rand requirementsUsed @ 0.3
    rand sEteamParticipatesInPlanning @ 0.3
    rand documentedRequirements @ 0.3
    rand sQAactivities @ 0.3
    rand (reviewRequirementChanges @ 0.3
          rany softwareConfigurationManagement@0.3
          rany baselineChangesControlled @ 0.3
          rany workProductsIdentified @ 0.3
          rany softwareTracking @ 0.3).
```

`Rors` is useful for specifying the high-level goals of the system. While `rany` will search *some subset* of its parameters, `rors` will search *all* its parameters. For example, in this less-than-perfect world, it is unlikely we can be *rich* and *happy* and *healthy*, but that should not stop us from trying; i.e.

```
goal if rich rors happy rors healthy
```

The operators `rand`, `ror`, `rors` and `rany` have the same satisficing criteria as `and`, `or`, `ors` and `any` respectively, but the latter search left-to-right while the former search in a randomly selected order. In terms of the `CombineRules` described above, `And` and `rand` are *conjunctive* operators; `Or` and `ror` are *simple disjunctive* operators; and the rest are *summing disjunctive* operators.

4. The CMM2 Model

CMM2 written in JANE has 55 `Item`s with `Value` $\in \{t, f\}$. Of the 55 `Item`s, 27 were identified as management actions that could be changed by managers (see Figure **??**). The top-level `Goal` was *goodProject*.

Within the model, `Chances` values were added to 79 edges of the 150 edges in the model. While each of these values are based on expert judgement, their precise value is subjective. Hence, each such `Chances` value X was altered to be a range

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

so the simulator could experiment with values nearby the stated `Chances` value.

The full model is available via email from the authors.

5. Learning From JANE

One execution of JANE can be summarized by the `Cost` and `Chances` of the top-level goal *goodProject*, the 79 values assigned to the subjective edges, and the values assigned to the 27 manager's actions. Random walking over JANE models can hence generate an overwhelming amount of data (79+27+2=108 data points per simulation, times the number of simulations).

Summarization of JANE output is performed by the TAR2 *treatment learner*. A treatment learner outputs a set of possible *treatments* (Δ) using the process described below. Each treatment is assessed by the *change* it makes to the distribution of classes. A *good* treatment increases the frequency of the *better classes* (where "better" is defined below).

Treatment learners input a set of classified examples (Eg). Each example is a conjunction of attribute ranges, plus one classification c. This classification comes from a pre-defined set of classes C; i.e. $c \in C$. Each classification c is associated with a numeric *score* i.e. some classifications score better than others.

For the CMM model, after 2000 random walks, a wide range of generated `Cost`s and `Change`s in *goodProject* were observed. These ranges were sub-divided into high/low bands of roughly the same size. Combing high/low `Cost`/`Chances` yields four classes. These classes were scored as follows:

Score=0: *High cost, low chance*
Score=1: *Low cost, low chance*
Score=2: *High cost, high chance*
Score=4: *Low cost, high chance*.

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

A treatment learner searches for *candidate* attribute ranges; i.e. ranges that are more common in the best classification than in the not-so-best classification. In the CMM domain, such a candidate is an attribute range that would tend to drive the system into *low cost, high chance projects*.

A heuristic ranking δ is given to each candidate reflecting (i) how much more common is the candidate in the best class than in some non-best-class c; and (ii) how much better is the best class than class c:

$$\delta = (score(best) - class(c)) * \left(\frac{|Eg \cap A = R \cap class = best|}{|Eg \cap A = R \cap class = c|} \right)$$

where $A = R$ denotes an attribute A with range R and $|Eg \cap X|$ denotes the size of the subset of examples that contain X.

Candidates with a large δ score are attribute ranges that are far more frequent in the best class (*high chances, low cost* projects) than in other classes. The *treatments* Δ are all subsets of candidates with a ranking higher that some user-specified δ threshold value δ_T.

In reality, it is hard to change many aspects of a project and the wise software engineer seeks the smallest number of changes with the greatest impact. Assuming we seek three changes, there are 560 possible treatments of size 3 in our 16 candidates. The *best* treatment is the one that offers the biggest improvement to the class distributions seen in the untreated data. To find the best treatment, we compute following ratio for all 560 treatments:

$$worth(\Delta_i) = \frac{\sum_{c \in C} |Eg \cap \Delta_i \cap class = c| * score(c)}{baseline\ worth} \quad (1)$$

where *baseline worth* is the *worth* seen in the untreated example set:

$$baseline\ worth = \sum_{c \in C} |Eg \cap class = c| * score(c)$$

6 Results

Figure 2 shows the three best treatments ($\Delta_1, \Delta_2, \Delta_3$) found using this technique (and Figure 1 compared the effects of these treatments to the untreated examples). Note that the values of each attribute are reported using the tags *no*, *lower*, *middle*, or *upper*. In treatment learning, continuous attribute ranges are divided into N-discrete bands based on percentile positions. For N=3, we can name the bands *lower*, *middle*, *upper* for the lower, middle, and upper 33% percentile bands.

In Figure 2, Δ_2 and Δ_3 are advising to lower the cost of:

Using requirements: This could be accomplished by (e.g.) by sharing them around the development team in some searchable hypertext format

Δ_1: *requirementsUsed.*`Cost`*=lower and not periodicSoftwareReviews and formalReviewsAtMilestones.*`Cost`*=lower*

Δ_2: *requirementsUsed.*`Cost`*=lower and goodUnitTesting.*`Cost`*=middle and formalReviewsAtMilestones.*`Cost`*=lower*

Δ_3: *goodUnitTesting.*`Cost`*=lower and periodicSoftwareReviews.*`Cost`*=middle and formalReviewsAtMilestones.*`Cost`*=lower*

Figure 2. The three best treatments found in the CMM-2 model.

Performing formal reviews at milestones: This could be accomplished by (e.g.) using ultra-lightweight formal methods such as proposed by Leveson [2].

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

The value *no* refers to missing values; a report of *X=no* is a recommendation *not* to use X as part of a treatment. Such a negative recommendation is seen in Δ_1 which is advising against *periodicSoftwareReviews* (plus lowering the costs of using requirements and formal reviews at milestones). Note that if *periodicSoftwareReviews* are conducted, Δ_3 is saying that there is no apparent need to reduce the cost of such reviews.

7. DISCUSSION

Space limitations prevent a review of work related to this research such as the goal-graphs of Chung et.al. [1] (such a review is available from the authors). This discussion section will hence focus on the major threat to the external validity of the above conclusions.

The knowledge source used in this study (CMM2) is widely, but not universally, accepted. Proponents of some other knowledge source could reject the specifics of the above conclusions, but still use the general technique; i.e. they could encode their preferred knowledge source in JANE and repeat the above analysis.

References

[1] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

[2] N. Leveson, S. Cha, and T. Shimall. Safety verification of ada programs using software fault trees. *IEEE Software*, 8(7):48–59, July 1991.

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