Applictions of Computational Intelligence to Quantitative Software Engineering

Tim Menzies

Dept. Electrical & Computer Eng. University of British Columbia,
2356 Main Mall, Vancouver, B.C. Canada, V6T 1Z4 <tim@menzies.com>

January 21, 2001

Most of software engineering research is not about most software engineering. Much of that research is funded by large organizations (e.g. DARPA) to support the needs of large software organizations. Large government organizations have the following features: projects last many years; funds are available for extensive, prolonged, and detailed analysis; stable long-term relationships exist between clients and consultants or contractors; there is no pressure for immediate delivery; and teams are very large. These features that are not reflected in the wider software engineering where most software is written by teams (less than a dozen people) and deadline pressure can be very tight. This mis-match between researched theory and general practice was starkly visible at the recent International Symposium on Software Reliability (San Jose, California, 2000). A keynote address from Sun Microsystems shocked the researchers in the audience: few of the techniques endorsed by the SE research community are being used in fast-moving dot-com software companies. For such projects, developers and managers lack the resources to conduct *heavyweight software modeling*; e.g. the construction of complete descriptions of the business model or the user requirements.

To better support the fast pace of modern software, we need a new generation of *lightweight software modeling* tools. Lightweight software models can be built in a hurry and so are more suitable for the fast-moving software companies. However, software models built in a hurry can contain incomplete and contradictory knowledge. For example, Figure 1 (left hand side) shows a lightweight model hurriedly assembled about `happiness` and `health` and `exercise`. This model is ambivalent on numerous details. Lines 13 and 14 make contradictory assumptions about `tranquil` and Lines 28-31 make contradictory assumptions about `likesSweat`. Since the model contains no rules to prove `tran-`

```
 1  domain= 'tutorial e.g. 1'.
 2  tim=    [month=jan,day=18,year=2000,
 3           elm='tim@menzies.com'].
 4  known=  [baseball ,content ,exercise ,
 5           football ,happy ,healthy
 6          ,likesSweat ,overweight ,rich ,
 7           running ,sick ,smoker ,swimming
 8          ,tranquil].
 9
10  tim says cost=0 and chances=1.
11  happy   if rich rors healthy rors content.
12
13  rich if not tranquil.
14  content if tranquil.
15
16  healthy if no sick.
17
18  tim says cost=0 and chances=1.
19  sick if smoker.
20  sick if overweight.
21
22  overweight if no exercise.
23
24  exercise if baseball rany running
25              rany swimming rany football.
26
27  tim says cost= 1 to ++4 and chances=0 to +1.
28  baseball if likesSweat.
29  running  if likesSweat.
30  football if likesSweat.
31  swimming if not likesSweat.
32
33  run :- prove(happy).
```

```
 1  ============================
 2  ** FINDINGS
 3  happy=t [cost=3.9626,chances=1]
 4
 5  ** REASONS
 6  content=t [cost=0,chances=1]
 7  happy=t [cost=3.9626,chances=1]
 8  healthy=t [cost=3.9626,chances=0]
 9  likesSweat=t [cost=0,chances=1]
10  overweight=t [cost=3.9626,chances=0]
11  running=t [cost=3.9626,chances=0.2225]
12  smoker=t [cost=0,chances=1]
13  tranquil=t [cost=0,chances=1]
14  no exercise=t [cost=3.9626,chances=0]
15  no sick=t [cost=3.9626,chances=0]
16
17  ============================
18  ** FINDINGS
19  happy=t [cost=1.54203,chances=1]
20
21  ** REASONS
22  happy=t [cost=1.54203,chances=1]
23  healthy=t [cost=1.54203,chances=0]
24  likesSweat=f [cost=0,chances=1]
25  overweight=t [cost=1.54203,chances=0]
26  rich=t [cost=0,chances=1]
27  smoker=t [cost=0,chances=1]
28  swimming=t [cost=1.54203,chances=0.1219]
29  tranquil=f [cost=0,chances=1]
30  no exercise=t [cost=1.54203,chances=0]
31  no sick=t [cost=1.54203,chances=0]
```

Figure 1: A lightweight model (left) and some outputs (right) generated using the techniques discussed in this paper.

`quil` or `likesSweat` so we might assume {`tranquil` and `likeSweat`} or {`tranquil` and `not likeSweat`} or {`not tranquil` and `likeSweat`} or {`not tranquil` and `not likeSweat`}. Also, the precise financial cost and chances of the different exercise regimes is not known to great detail. The construct `cost= 1 to ++4` at line 27 means that these these sports can cost between one and four units each, with the mean leaning heavily towards four. The odds that this author will play sports are `cost=0 to +1` (see line 27). This means that that the chances of me playing sports are between 0 and 1, with the mean leaning slightly towards one. The other unique features of the language used in Figure 1 will be discussed below.

The presence of uncertain and contradictory knowledge in lightweight models complicates their processing. Suppose some inference engine is trying to build a proof tree across a lightweight software model containing contradictions. Gabow et.al. [4] showed that building pathways across programs with contradictions is *NP-hard* for all but the simplest software models (a software model is very simple if it is very small, or it is a simple tree, or it has a dependency networks with out-degree $\leq$ 1). No fast and complete algorithm for NP-hard tasks has been discovered, despite decades of research.

Empirical results offers new hope for the practicality of NP-hard tasks such as reasoning about lightweight models like Figure 1. A repeated and robust empirical result from the satisfiability community (e.g. [12, 1]) is that theoretically slow NP-hard tasks are only truly slow in a narrow *phase transition* zone between under-constrained and over-constrained problems. Further, it has been shown empirically that in both the under/over-constrained zones, seemingly naive randomized search algorithms execute faster than, and nearly as completely, as traditional, slower, complete algorithms [12]. It is easy to see why this might be so. In the over-constrained zones, it is impossible to satisfy all constraints so we need not search very long. In the under-constrained zone, many solutions exist so, once again, we need not search very long. In related research:

- A literature review by Menzies and Cukic found numerous cases in the SE and knowledge engineering literature where a small number of random probes into a system yielded as much information as a much larger number of probes [8].

- Menzies et.al. developed a general mathematical model of random abductive search which, on simulation, found that a small number of random probes into a system usually yielded as much information as a much larger number of probes [9, 10].

These empirical results suggest that we might be able to simply the exploration of Figure 1 using randomized search. Note that what we can't do is simply translate the results from the satisfiability community into SE. The predictors for the phase transition zone are expressed in a form that is too low-level for the average software engineer; e.g. Selman's linear clause model does not refer to design structures that the average software engineer would recognize [12]. Recent results suggest that the available predictors for the phase transition zone are incomplete [5]. We have some preliminary results strongly suggesting that we can get better and more detailed predictors for the phase transition zone by assuming theories are expressed as *horn clauses* and not the conjunctive normal form using by the satisfiability community [10]. For example, these results can compute a mathematical probability that randomized search will be an adequate search strategy for particular systems. Computing this probability will be essential if randomized search is to be applied to safety-critical software.

My research hence focuses on *random abductive search over horn clause theories*. In many domains, software engineers can understand *horn clause* representations of their models. For example, database modelers find it easy to map from SQL into the horn clauses of Prolog. Also, one way to formalize Figure 1 is to express the rules as horn clauses.

Abduction is computational intelligence procedure that is a natural method of processing theories containing contradictions. When abduction finds a contradiction, it forks one world of belief for each possibility. Each world *fixes* the uncertainties in a theory by committing to a particular set of consistent assumptions [7]. *Randomized abduction* explores a small number of randomly selected fixes.

For example, Figure 1 (right hand side) shows two random abductions over our lightweight models. To explain this figure, we have to explain the inference procedures use in that figure.

- Every `known` term has a default `cost` and `chances` of 0 and 1 respectively.
- Rules infer terms and rules have a head and a body, separated by the `if` keyword.
- Our belief in a rule head are `chances` times our belief in its body and the cost of believing a rule head is the cost of believing in its body, plus `cost`.
- Terms that appear in no rule head are assumptions (e.g. `tranquil` and `likesSweat`). Once an assumption is made, all contradictory assumptions are forbidden; e.g if we assume `likesSweat` then we must not later assume `not likesSweat`.
- Rule bodies are connected by the standard operators `and, or, not` and by some random search operators such as `rors, no, rany`.
- `Rors(`$X_1, X_2, ...$`)` first randomly shuffles the order of X_i, then tries to prove them in the shuffled order. If any X_i is proved, then subsequent proofs must be consistent with those prior assumptions. If any X_j is disproved, then `rors` moves on to try and prove X_k ($k > j$).
- `Rany(`$X_1, X_2, ...$`)` acts like `Rors` expect that it runs on a subset of X_i (subset chosen at random).
- `No(`X`)` collects all the evidence for X and subtracts that from one.

By running the above procedure N times, an analyst can sample the range of possibilities within a lightweight theory. Two such runs are shown in Figure 1 (right). Note that we have found two different ways to prove `happy`ness (at lines 3 and 19) but with very different costs. By sorting all these runs on increasing cost and decreasing chances, we can find the high chance low cost methods of achieving our goals. In this example, we can be happy with a cost of 1.54 (line 19) if we go `swimming`. However, if we go `running` instead, `happy`ness will cost us 3.96 (line 3).

Related Work

Abduction has been extensively studied elsewhere and applied to fields such as model-based diagnosis [7].

The general problem of reasoning about inconsistent SE models has been well studied. For example, in the SE literature, requirements engineering researchers have explored conflicting requirements generated either from non-functional requirements (e.g. [11]) or from multiple stake holders (e.g. [2, 3]). A standard technique for this exploration is some contradiction tolerant logic such as the LTMS [11] or the quasi-consistent logics [6]. This research is an attempt to simplify the standard technique. In systems where random search can adequately probe a space of uncertainties, then very simple inference techniques will suffice to probe the space of "maybes". If we can demonstrate that randomized search is widely useful, then we can design a new generation of very simple contradiction tolerant reasoners.

Discussion

The above case study is a small study where a few random abductive adequately explored a theory and found cost-effective strategies for achieving `happy`ness. The open research issue is this: *for what domains can we guarantee that a practical number of runs will suffice for sampling the behaviors of a theory?*

If we can build a general predictor for where fast random search will suffice for lightweight software modeling, then we could better support the fast-pace of modern SE. For example:

- Suppose we could *define lightweight design principles* that always lead to software models that can be quickly and adequately probed via randomized search. For those systems, we can quickly discover the implications of the uncertainties within our lightweight software models.

- Also, we could use our *randomized abductive theorem provers* to optimize tasks such as generating test cases from specifications, diagnosing the cause of faulty outputs, or understanding the consequences of conflicting requirements from different stake holders.

- Further, we could *define early stopping rules* for the testing a specification. In systems where fast random probing will suffice, a small number of random probes will reveal most of what we will reach via a much larger number of probes.

References

[1] P. Cheeseman, B. Kanefsky, and W.M. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–337, 1991.

[2] S. Easterbrook. Handling conflicts between domain descriptions with computer-supported negotiation. *Knowledge Acquisition*, 3:255–289, 1991.

[3] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh. Inconsistency handling in multi-perspective specification. *IEEE Transactions on Software Engineering*, 20(8):569–578, 1994.

[4] H.N. Gabow, S.N. Maheshwari, and L. Osterweil. On two problems in the generation of program test paths. *IEEE Trans. Software Engrg*, SE-2:227–231, 1976.

[5] H.H. Hoos and C. Boutilier. Solving combinatorial auctions using stochastic local search. In *Proc. of AAAI-2000*, pages 22–29. MIT Press, 2000.

[6] A. Hunter and B. Nuseibeh. Analysing inconsistent specifications. In *International Symposium on Requirements Engineering*, pages 78–86, 1997.

[7] A.C. Kakas, R.A. Kowalski, and F. Toni. The role of abduction in logic programming. In C.J. Hogger D.M. Gabbay and J.A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming 5*, pages 235–324. Oxford University Press, 1998.

[8] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. To appear. Available from `http://tim.menzies.com/pdf/00ijait.pdf`.

[9] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, 2000. Available from `http://tim.menzies.com/pdf/00iesoft.pdf`.

[10] Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing nondeterminate systems. In *ISSRE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00issre.pdf`.

[11] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions of Software Engineering*, 18(6):483–497, June 1992.

[12] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI '92*, pages 440–446, 1992.