LEAN = (LURCH+TAR3) = Reusable Modeling Tools

Tom Burkleaux, Tim Menzies
Computer Science, Portland State University,
Oregon, USA
tomb@wirelily.com|tim@menzies.us

David Owen
Computer Science and Electrical Engineering
West Virginia University, Morgantown, WV USA
owen@freeshell.org

June 26, 2004

Abstract

Declarative knowledge describes facts and relationships within a domain. In theory, declarative knowledge can be easier to understand that procedural knowledge; eg. it can be easier to modify; easier to communicate to others; and easier to reuse for different purposes.

This paper tests that theory. After building a formal model of a commercial factory, we asked what else can we do with this model? Originally, the intent was to just to simulate the factory. Subsequently, we found that the simulation was useful for optimization and fault localization *without* additional architecture.

1 Introduction

Software Engineering (SE) has traditionally been structured around a series of sequential analytical procedures that lead towards a solution to the problem at hand; e.g. the waterfall model. A significant shortcoming with that approach is the inability to efficiently explore and evaluate a wide range of options. It has been claimed that the goal of SE is "...not in finding *the* solution, but rather, in engineering an *acceptable* or *near optimal solution* from a large number of alternatives. [2]"

Clarke et al [2] suggest the use of "metaheuristics techniques" to improve SE. They propose that many SE tasks are really search problems. While search has been used for tasks such as model checking, test data generation, and module clustering, they suggest that search can be applied to a wider variety of SE tasks such as maintenance/evolution, system integeration, and requirements scheduling. An essential precondition for their metaheuristic methods is that numerous candidate solutions can be quickly generated-something that can't be guarenteed in every domain.

Like Clarke et al, Menzies has previously [8, 9] argued for a unification of common knowledge engineering (KE) practices. He claimed that a wide range of knowledge engineering tasks can be mapped to a single abductive procedure, using a range of different selection criteria. Menzies' approach hit the same stumbling block as the metaheuristic method: the need to rapidly generate candidate solutions. Abduction can be very slow (theoretically, it is NP-hard [1] and, empirically, runs in time exponential on theory size [8]).

Stochastic methods are known to be rapid; e.g. see the studies with stochastic search in satisfiability [19] and planning [6]. This paper began with the question: can stochastic methods enable the Clarke/Menzies approach? The result was LEAN: a novel combination of Clarke et.al's metaheuristic search, Menzies' abduction, and stochastic *serial world generation* (described below).

In the case study described below, LEAN is used first as a simulation tool, then as a optimizer, then as a fault localization tool. In a result that is encouraging for the Clarke/Menzies goal of simpler SE/KE, no change to the LEAN architecture was required to achieve these three tasks.

2 Abduction

Informally, abduction is typically defined as inference to the best explanation (e.g. [20]). Given α , β, and the rule $R_1 : \alpha \vdash \beta$, then:

- *deduction* is using the rule and its preconditions to make a conclusion; i.e. $\alpha \wedge R_1 \Rightarrow \beta$;
- *induction* is learning R_1 after seeing numerous examples of β and α;
- *abduction* is using the postcondition and the rule to assume that the precondition could explain the postcondition; i.e. $\beta \wedge R_1 \Rightarrow \alpha$ [7].

More formally, abduction is the search for assumptions A which, when combined with some theory T, achieves some set of goals G without causing some contradiction [3]. That is $T \cup A \vdash G$ and $T \cup A \not\vdash \perp$. In the general case, multiple *worlds of consistent belief* might be generated.

Abduction is not a certain inference and its results must be checked by an inference assessment operator (which we call $BEST$ and Bylander et.al. [1] call the plausibility operator *pl*). Formally, $BEST$ selects some subset of the generated worlds. In the experiment described below, $BEST$ is implemented by a utility function that scores runs across our model.

While abduction can be used to generate explanation engines (see below), we believe that abduction is more than just a description of "inference to the best explanation." Abduction can be summarized as follows: make what inferences you can that are relevant to some goal, without causing any contradictions. A variety of knowledge-level tasks can be implemented via an appropriate selection of $BEST$ assessment operators [8, 9].

2.1 Stochastic Serial World Generation

Abduction generates consistent *worlds of belief.* These worlds can be generated in parallel as the abductive inference engine searches for some goals. That is, fork one world for each possibility whenever a conflict is encountered and recurse the abductive device on each new world. This can be a very slow process [1,8]. For example N binary conflicts can generate up to 2^N worlds.

An alternatve to parallel world generation is serial world generation. In one run, when N choices are found, a serial world generator could pick one at random and ignore the rest. Note that, by definition, this generates one randomly selected consistent world of belief. After some stopping criteria is reached (e.g. depth of search), the serial abductive device could reset and start again from the intial conditions to generate another randomly selected world of belief

Two such stochastic serial world generators are HT0 [16] and LURCH [17, 21–24]. HT0 executes over horn clauses while the LURCH tool used in this paper executes over finite state machines.

HT0/LURCH are stochastic and, like any other heuristic method, are incomplete. The advantage of serial world geneation is that the memory requirements are linear on the number of resets while the paralled method can consume exponential memory before any world reaches any goal. Empirically, HT0/LURCH have been observed to scale to theories too large for complete search.

Further, HT0/LURCH are simple to implement (not much more complex that a depth-first search) and can run much faster than more complete methods. For example, the median time for LURCH and NuSMV to find a single error in one large flight guidance system was 3 minutes and 125 minutes (respectively) [24].

3 LEAN

LEAN is an application of stochastic serial world generation. LEAN assumes that within a complex space of options interrelated by influences and constraints, there are a small number of variables that control whether a particular class of options are better than another class, and that these variables can be learned through simple methods.

LEAN is built upon two lower-level tools: LURCH and the TAR3 treatment learner [4, 5, 10–15]. LURCH generates sample behaviors from a formal model and TAR3 explores those samples for transitions that tend to select to preferred solutions.

To implement LEAN, we added to LURCH two features: a *utility* function and a *nudge* function:

- The utlity function that scores each run, or world generated. Options cannot be judged vis-a-vis other options without some conception of what defines a better choice. To that end, the user must provide a scoring function that rates the utility or fitness of each world generated.

- The nudge function changes the probability that LURCH will select a certain transistion to fire. LURCH's default behavior is to select transistions assuming that they all have the same probability. The nudge function is an API that lets another program skew that selection process.

Each LEAN runs collects counts of transitions within that run, and the utility score. After generating a large number of worlds – the default setting is 500 – the transition count and utlity score for each run are analyzed by TAR3, a treatment learner. The runs are separated into a small number of classes by discretizing the utility scores. The classes are ranked from best to worst. TAR3 then searches for the smallest set of "treatments" that define a higher-value class in comparison to a lower-value class [10]. These treatments offer settings to the nudge function that results in the next run of LEAN favoring transistions associated with higher-value classes.

The *LEAN cycle* consists of randomly generating a large number of worlds which are scored with a utility function, then learning the transitions associated with better scores. The statistics of the average count for each transition, combined with a treatment involving that transtion, indicated whether the bias for that transition should be adjusted up (or down).

The results of each run of LEAN are fed back into the transition biases. The system learns what transitions to favor in order to improve the overall average utility score of the worlds generated. The favored transitions represent better choices or policies that can be implemented.

4 Case Study

To test LEAN, we setup a model for examining the impact of equipment choices for a small-scale vodka distillery. The production of vodka involves a number of different phases, and each phase can be accomplished in a number of ways. The overall flow of the process goes mash–beerstriping–distilling–filter–water–bottling.

Each run of LEAN randomly chooses a piece of equipment/method from each option-set, and then executes the model to to determine the utility of that particular combination (see Figure 1).

4.1 Utility Function

The utility of any combination of equipment is a function of four variables: quality, labor, revenue, and setup costs. These inputs are weighted so each input has a relatively equal influence. Utility is a numeric score, with quality and revenue increasing the score, and labor and setup costs descreasing the score.

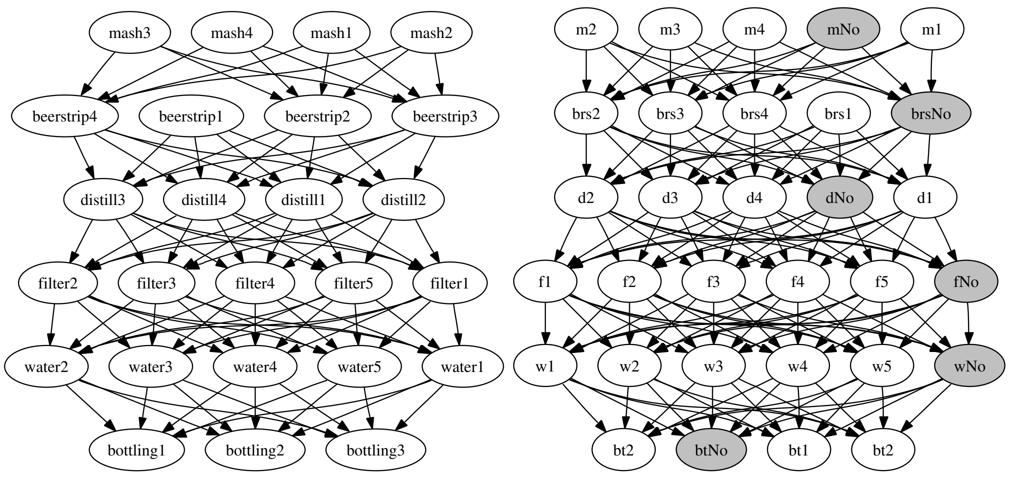


Figure 1: Space of equipment options, normal model (left), faulted model (right). The faulted model is identical the the normal model with the addition of the faulted choices (grey)

The utility function was developed in discussions with a financial stakeholder in the vodka distillery, and represented the utility the stakeholder wishes. This utility function was choosen with the hope that it will elicit policies within the space of options that will achieve better utility, as judged by the domain expert.

4.2 Representation

Our model of the distillery has two logical levels, and could be conceived of as two separate finite state machines. The first level was a state machine for deciding the configuration of the distillery by choosing the equipment. LURCH randomly chooses the configuration on each run, and these choices are a policy. The space of options for choosing equipment is shown in Figure 1.

The second level is a state machine representing the current configuration. Once the equipment is choosen, this machine representing the actual equipment choosen is executed. As the machine executes, global variables representing the quantity of product produced, along with auxilliary information such as accumulated quality, cost, and labor, are updated. The utility function is calculated from runs across this second machine.

The state machine representing the operation of the equipment contains all possible choices, with each guarded by the equivalent of a boolean, which is decided by the first state machine. We choose this structure because our utility function could not calculated from the policy alone, i.e. particular set of equipment decisions.

For this representation, we setup LEAN to learn off the choices that make up a particular policy. This means we were looking for the best combination of equipment choices.

5 Experiment 1: Simulation

After setting up our model, we ran the model through LEAN. The results of the first run can be seen in Figure 2, upper left. Each point in our graph represents one run of LEAN, that is, a set of decisions that built a distillery, and an execution of that setup which is scored.

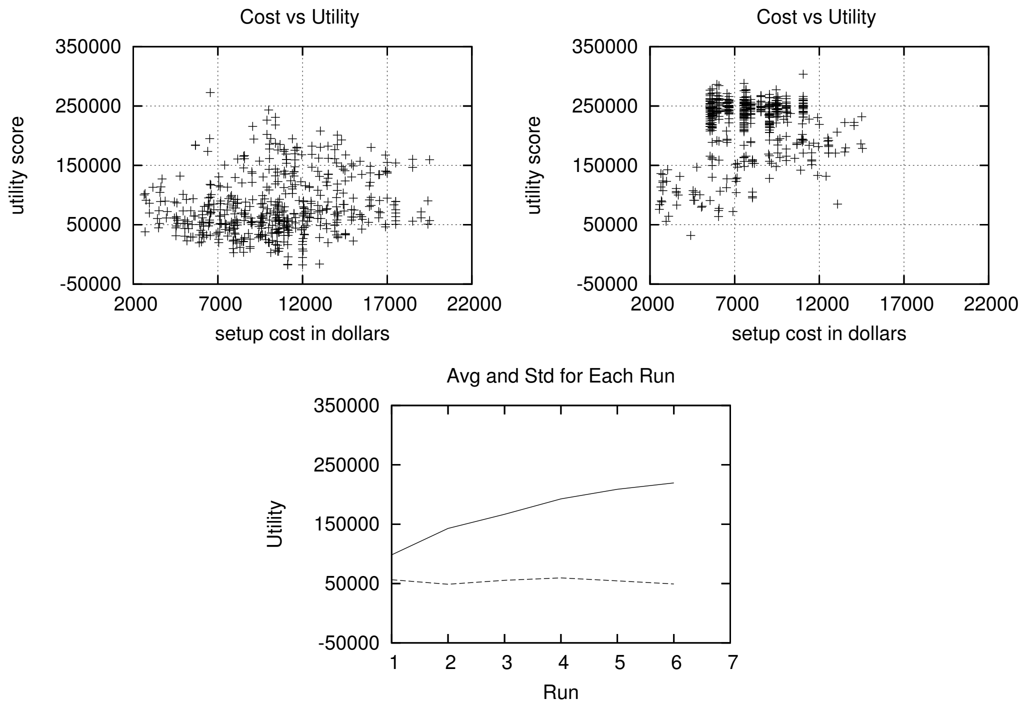


Figure 2: Normal Model: Run 1, Run 7, and Change in Utility

There is an even distribution of the plot of the setup cost versus the utility score. These results are from random executions of our model without any learning. Thus they represent the untreated model, and serve as our baseline.

6 Experiment 2: Optimization

We took our baseline model ran it through LEAN. Each run is composed of the following stages:

First the LURCH engine randomly executes the model for 500 iterations. The random-ness of the next transition choosen is affected by the bias value associated with that transition. For the untreated model, all the biases are equal. After each iteration, the count of transitions and the utility score are collected.

Next, the transitions and scores are broken into classes for treatment learning. Then they are feed to the TAR3 treatment learning engine. TAR3 produced a number of treatments indicating choices that would produce the greatest lift of the lower-valued classes to the higher-valued class.

Finally, treatments (from the top ten treatments) that indicated a clear preference for a decision transition were used to increase the bias of that transition prior to the next iteration.

For our experiments, we used a simple nudge function. Associated with each transition was a bias value, which defaults to 1. At each step in the execution of the model, all currently allowable transitions are placed into a sorted list, based on a random number between 0 and 1. After all allowable transitions are identified, the transition with the lowest number is choosen, i.e. first in the queue. When the bias for a transition is greater than 1, before the transition is place in the queue, the transition gets the lessor of x random numbers, where x = bias. This increases the probability the transition will be at the front of the queue, and thus choosen. For negative biases, a similiar procedure is used, except that the greater of x random numbers is choosen.

For example, if t1 was a decision transition, and the treatment indicated this decision should be baseline

utility	cost $2000+	$7000+	$12000+	$17000+	tpcts
250k +	0	0	0	0	0%
150k +	0	7	4	1	12%
50k +	12	26	18	4	60%
-50k +	5	17	6	0	28%
tpcts	17%	40%	28%	5%	100%

after run 7

cost utility	cost $2000+	$7000+	$12000+	$17000+	tpcts
250k +	12	17	0	0	29%
150k +	18	38	4	0	60%
50k +	8	3	0	0	11%
-50k +	0	0	0	0	0%
tpcts	38%	58%	4%	0%	100%

Figure 3: Cost vs Benefits, By Pct for the normal model. Baseline shown left, distribution after 7 runs shown right. Each cell shows the percentage of the runs that generated utilities falling into that cell.

choosen, then the bias of t1 would be increased by 1. On the next iteration, LEAN uses the accumulated biases from the previous runs.

After seven runs of LEAN, we achieved the results shown in Figure 2, top right. LEAN learned which transitions to bias in order to both improve the average utility and lower the average cost. This improvement in the output of the model, from lower-score and high setup cost to higher-score and lower-cost is represented numerically in Figure 3. Note that, by run 7, the mean utility had increased and the mean cost had decreased.

The application of LEAN on our basic model resulted in only three transitions having their biases increased. This supports the hypothesis that a small number of transitions have a large impact on the overall behavior of systems.

6.1 Business Implications

The more important question to ask of LEAN is what support it gives to decision making. After running LEAN on the model, we have a set of policy recommendations – these are the transitions with the highest bias. In the case of our basic model, three transactions had their biases increased.

Each of these three transitions were one choice from their level of the decision machine. For the current model and utility function, LEAN gave us three critical choices we should make to improve the overall score. The critical choices recommended were:

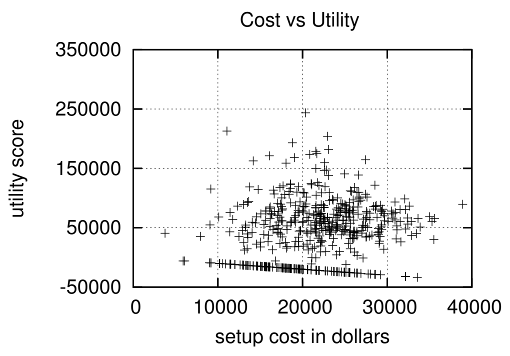
{ *beerstrip1, distill4, and filter4* }

This raises the question of whether the other decisions have any impact at all? We believe that LEAN gave us the choices with the highest impact. That is, these choices probably have the highest order of impact on the behavior of the overall system.

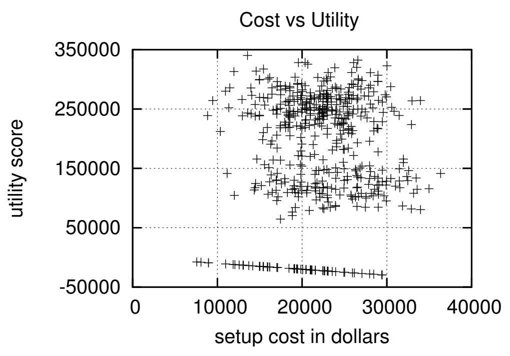
The improvement in the average utility of the option space can be seen in Figure 2, bottom graph. By run 7, the average utility has increased from approximately 80,000 to 220,000.

7 Experiment 3: Fault Localization

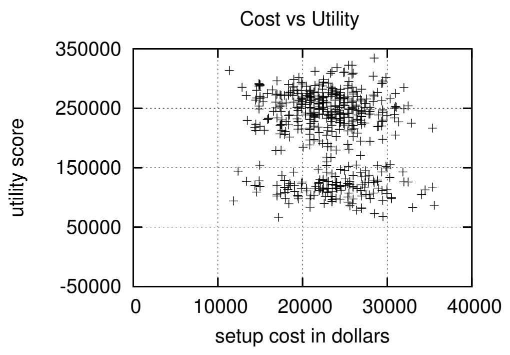
For a follow-up experiment, we wanted to see what if LEAN-ing was useful for finding faults in our distillery. To generate the faulty model we took the distillery model we used in our initial experiment and seeded it with "faults". For each equipment set, we included a choice of "no choice". This makes it possible to set up a distillery with vital pieces of equipment missing. For a graph of this space of options see Figure 1. This space has the same information of the non-faulted model, with the addition of the extra possibility of "faulted" choices.



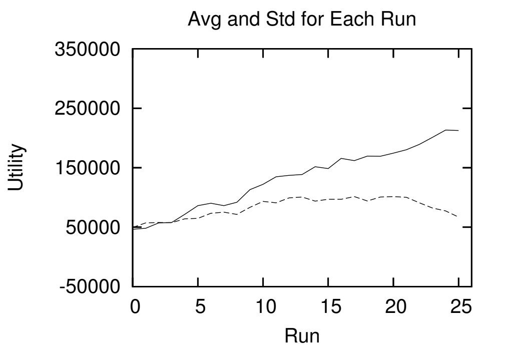
Run 1: Baseline



Run 22: Before Fault Identification



Run 26: After Fault Identification



Runs 1-26: Change in Avg. Utility

Figure 4: Faulted Model

The results from the initial run of the faulted model as shown in Figure 4, upper-left. The distribution of scores for the faulted model exhibits clustering into two distinct regions.

We ran LEAN on the faulted model for 22 runs. The results of this are shown in Figure 4, upper-right. There are two items to note. First, there was improvement in the utility score for *one* of the clusters. The top cluster of scores from the first model showed improvement. Second, however, the cluster of points along the diagonal line has shown little improvement.

The question was raised why it appears LEAN didn't learn to avoid the "faulted" choices?

The two clusters showing in both the upper-left and upper-right of Figure 4 suggest that the clusters represent two classes and further suggest that treatment learning should be able to find the "treatment" that lifts one class to the other.

7.1 Finding Faults

To find this fault, we reversed the direction of the treatment learning and tried to treat the upper cluster and move it towards what we assumed was the faulted cluster. We were given a number of treatments to use which we used to build negative biases.

After four further runs using this diagnostic procedure, we achieved the result shown in the lower-left of Figure 4. The "faulted" cluster has disappeared almost entirely.

Figure 5 shows the final results. In our baseline run of the faulted model, 97% of the utility scores fell below 150,000, and only 3% of scores were above 150,000. By run 26, only 27% of utility scores were below 150,000, with 73% of the scores now above 150,000.

Once we "diagnosed" the faulty region, and setup transitions to avoid (versus just prefer relative to baseline

	cost				
utility	$2000+	$7000+	$12000+	$17000+	tpcts
250k +	0	0	0	0	0%
150k +	0	1	2	0	3%
50k +	0	13	29	4	46%
-50k +	2	23	24	2	51%
tpcts	2%	37%	55%	6%	100%

after 26 runs

	cost				
utility	$2000+	$7000+	$12000+	$17000+	tpcts
250k +	0	12	25	1	38%
150k +	0	8	26	1	35%
50k +	0	6	19	2	27%
-50k +	0	0	0	0	0%
tpcts	0%	26%	70%	4%	100%

Figure 5: Faulted Model, By Pct. Baseline shown left; distribution after 26 runs shown right. Each cell shows the percentage of the runs that generated utilities falling into that cell.

another transition), the standard deviation decreased. This can be seen from runs 23 to 26 in bottom right of Figure 4.

7.2 Business Implications

What policies does LEAN recommend after operating on the faulted model? The recommendations provided by LEAN at run 22, before we attempted "diagnosis" on the discontinuos area of the graph, are as follows:

Yes:{ *beerstrip1, distill3(slight),distill4, filter4, water4(slight)* }
No:{ *filter1,filter2,filter3, filter5,water1, water5*}

The "no's" represent a negative bias away from that choice. The strong "yes" recommendations are exactly the same recommendations made for the normal (non-faulted) model.

After running diagnosis on the "faulted" region of the graph, the recommendations listed above were the same. But by run 26 the following "avoidance" recommendations were added:

Avoid Yes: { *beerstrip2* }
Avoid No: { *beerstrip1,distill4, water2,water4,bottle1,bottle2, bottle3* }

Again, these recommendation support the original recommendations. For example, the positive recommendation to choose {*beerstrip1 - yes*}, it backed up by the recommendation to avoid choosing {*beerstrip1 - No(avoid)*}.

The average utility of our option space increased from approximately 46,000 to 212,000 (see Figure 4, lower-right).

8 Discussion

This experiment shows how a tool like LEAN can be applied to software engineering tasks. This supports the assertion that metaheuritics have wide applicability as a SE method [2].

LEAN is high-level inference tool built upon LURCH, a stochastic model checker. Because LURCH uses random search, this brings into question whether the results from LEAN are incomplete or "quirky". Research suggest that random model checkers do lead to stable results [18]. It has been proposed that the internal states of systems "clump", and sampling within this space can retrieve within a small degree or error the same information as a complete search.

One advantage of random based search tools over complete search, is the ability to explore larger system. Compared to complete search, this translates into faster search of systems of the same size, and the ability to search systems that are beyond the practical size of complete search. And in SE situations where requirements, and thus design, change rapidly, random search has the ability to give valuable guidence at the early stages of design.

The execution time for each run of LEAN averaged 35 seconds of real user time – 19 seconds for the Lurch phase and 16 seconds for the Tar3 phase. Because LEAN is based on random sampling of the option space, we expect the execution time to increase linearly with the number of transitions in the model. After only 7 runs, our "optimization" experiment exhibited considerable improvement in average utility. This suggest LEAN should be able to handle large models with reasonable execution times.

Another question about search techniques as metaheuristics for SE is the need to define utility or fitness functions to guide the search. In the experiment we conducted, the results can only be understood as improvement the space of options with respect to the utility function. The usefulness of the results are directly dependent upon the questions we ask of our systems.

We are encouraged that the both our normal model and faulted model, when using the same utility function, produced the same set of policy recommendations. This suggest there is some underlying stability to the model in question.

As has been mentioned by others [2] our task should not viewed as producing the optimal solution, but in providing a range of good enough options. The combination in LEAN of LURCH and TAR3 appears to fulfill this goal. We did not determine an absolute best solution, but were able to achieve relative improvement from a baseline.

9 Future Work

We want to continue work on developing LEAN. Our primary focus will be on better nudge functions. Instead of the naive bias method we used for this experiment, we want to associate beta and gamma values for each transition and use these to determine their probability. This will allow us more sophisticated control over how we "nudge" our transition biases.

Another area to be explored are the policies of LEAN-ing. Basic questions to answer are: How rapidly to LEAN towards a favored transition? Can the topology of the state-space be used to improve the results? To what degree do random based methods miss information about the state space?

To confirm its usefulness in real-world projects, LEAN needs to be applied to wider range of complex models. And within the world of meteheuristic techinques, we want to compare LEAN to other search methods, such as simulated annealing, applied to the same task.

Finally, a future area we are interested in, is developing techinques for discretizing continuous model so tools such as LEAN can be applied. A major question in this area is loss of information during the discretization process.

10 Conclusion

We demonstrated the use of search as a software engineering tool for two tasks. First, exploring a space of options and discovering critical variables that lead to "better" designs. Second, our metaheuritic technique allowed us to identify and locate "fault" conditions within our space of options. Each of these two tasks lead to policies that could be implemented. Further, we demonstrated that LEAN is a practical metaheuristic tool. With the same architecture, we will able to apply LEAN to a number of software engineering tasks.

This also demonstrated the use of an abduction procedure as a tool for performing a knowlege engineering task. In this case, the KE task of optimization was performed on our model. The choice of our utility function – the $BEST$ operator – allowed LEAN to infer the transition biases that selected better worlds. The result of this inference was a set of policy choices that lead to a better option space. Using the same abduction procedure, with a different $BEST$ operator, we were able to find the location of faults in our model. This supports the propostion that common knowledge engineering (KE) practices can all be mapped to abduction.

For both abduction and the metaheuristic approach to software engineering, the stochastic search utilized by LEAN provides the ability to rapidly generate candidate solutions. This overcomes the stumbling block to their application as software/knowlege engineering tools. While the random search utilized by LEAN is incomplete, we have demonstrated the ability of LEAN to provide useful results over a number of tasks such as optimization and fault location.

References

[1] T. Bylander, D. Allemang, M. Tanner, and J. Josephson. The Computational Complexity of Abduction. *Artificial Intelligence*, 49:25–60, 1991.

[2] J. Clarke, J. J. Dolado, M. Harman, R. M. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings on Software*, 150(3):161–175, 2003. Available from `http://www.brunel.ac.uk/~csstrmh/papers/sbse.ps`.

[3] K. Eshghi. A Tractable Class of Abductive Problems. In *IJCAI '93*, volume 1, pages 3–8, 1993.

[4] M. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002. Available from `http://menzies.us/pdf/02re02.pdf`.

[5] Y. Hu. Treatment learning: Implementation and application, 2003. Masters Thesis, Department of Electrical Engineering, University of British Columbia.

[6] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, Aug. 4–8 1996. AAAI Press / MIT Press. Available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.

[7] H. Levesque. A Knowledge-Level Account of Abduction (Preliminary Version). In *IJCAI '89*, volume 2, pages 1061–1067, 1989.

[8] T. Menzies. *Principles for Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales, 1995. Ph.D. thesis. Available from `http://menzies.us/pdf/95thesis.pdf`.

[9] T. Menzies. Applications of abduction: Knowledge level modeling. *International Journal of Human Computer Studies*, 45:305–355, 1996. Available from `http://menzies.us/pdf/96abkl.pdf`.

[10] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In T. M. Khoshgoftaar, editor, *Software Engineering with Computational Intelligence*. Kluwer, 2003. Available from `http://menzies.us/pdf/02itar2.pdf`.

[11] T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://menzies.us/pdf/01lesstalk.pdf`.

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

[13] 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://menzies.us/pdf/01agents.pdf`.

[14] T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.

[15] T. Menzies and Y. Hu. Just enough learning (of association rules): The TAR2 treatment learner. In *Artificial Intelligence Review (to appear)*, 2004. Available from `http://menzies.us/pdf/02tar2.pdf`.

[16] T. Menzies and C. Michael. Fewer slices of pie: Optimising mutation testing via abduction. In *SEKE '99, June 17-19, Kaiserslautern, Germany. Available from `http://menzies.us/pdf/99seke.pdf`*, 1999.

[17] T. Menzies, D. Owen, and B. Cukic. Saturation effects in testing of formal models. In *ISSRE 2002*, 2002. Available from `http://menzies.us/pdf/02sat.pdf`.

[18] T. Menzies, D. Owen, M. Heimdahl, J. Gao, and B. Cukic. Nondeterminism: Unsafe?, 2003.

[19] D. G. Mitchell, B. Selman, and H. J. Levesque. Hard and easy distributions for SAT problems. In P. Rosenbloom and P. Szolovits, editors, *Proceedings of the Tenth National Conference on Artificial Intelligence*, pages 459–465, Menlo Park, California, 1992. AAAI Press. Available from http://www.citeseer.ist.psu.edu/mitchell92hard.html.

[20] P. O'Rourke. Working notes of the 1990 spring symposium on automated abduction. Technical Report 90-32, University of California, Irvine, CA., 1990. September 27, 1990.

[21] D. Owen and T. Menzies. Random search of and-or graphs representing finite-state models. In *Proceedings of the First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://menzies.us/pdf/01randandor.pdf`.

[22] D. Owen and T. Menzies. Lurch: a lightweight alternative to model checking. In *SEKE '03*, 2003. Available from `http://menzies.us/pdf/03lurch.pdf`.

[23] D. Owen, T. Menzies, and B. Cukic. What makes finite-state models more (or less) testable? In *IEEE Conference on Automated Software Engineering (ASE '02)*, 2002. Available from `http://menzies.us/pdf/02moretest.pdf`.

[24] D. Owen, T. Menzies, M. Heimdahl, and J. Gao. On the advantages of approximate vs. complete verification: Bigger models, faster, less memory, usually accurate. In *IEEE NASE SEW 2003*, 2003. Available from `http://menzies.us/pdf/03lurchc.pdf`.