Journal of Automated Software Engineering (submitted), WP:alp2/ase(May 2, 2000).

Some Empirical Automated Software Engineering

Tim Menzies[‡], Sam Waugh[†]

[‡]NASA/WVU Software Research Lab, 100 University Drive, Fairmont, USA; tim@menzies.com

[†]Defence Science and Technology Organisation, Air Operations Division, Melbourne, Australia; sam.waugh@dsto.defence.gov.au

May 2, 2000

Abstract

We experiment with core processes within software development tasks (reuse and conflict management in requirements engineering). After defining those processes, we automatically perform 100,000s of manipulations in accordance with that process. The results of the manipulations are summarized and discussed. In two case studies with this technique we find that building widely reusable software components will be harder than we had thought while resolving conflicts amongst competing stakeholders is easier than we had thought.

1 Introduction

Can software engineering be an exact science? Software engineering is a human-intensive process. Such a sociological-intensive exercise can forbid exact conclusions. Nevertheless, in some circumstances, exact conclusions are possible; e.g.

- Suppose human idiosyncracises are the main controlling factor in software development. If so, then we might predict that exact software cost estimation requires complex equations about human dynamics.
- Yet in at least one case study, this was not true. Chulani, Boehm, and Steece used a simple model of software costs and merely two dozen input parameters to produce reasonably accurate estimates of software construction time (their estimates were within 25% of actual, 69% of the time) [Chulani, Boehm & Steece 1999].

This article seeks other conclusions that are not effected by the idiosyncracises of human software engineers. We assume that when people write software, they are *manipulating theories*. As we hope to show, certain theory properties are *stable* across a wide range of theory manipulations. That is, fortunately, some of the properties of our theories may not be as idiosyncratic as the people who write them.

To make this case, we will perform manipulations on a general class of theories (and-or graphs with ground nodes). Since there are many possible manipulations on a theory, we will use an automatic rig to perform millions of manipulations such as injecting errors into theories; changing theory connections; alter the data sets given to a theory and alter the processing of a theory including totally random selection amongst alternatives. In the experiments reported here, we found that certain properties will emerged as stable, despite widespread manipulations. In particular, some tasks were much harder that expected, while others were much easier:

Much harder: Software reuse.

Much easier: Handling inconsistencies in requirements.

The rest of this article is structured is followed. §2 offers an example of our type of experimentation. As we define our experiments, we will encounter certain restrictive **suppositions** that threaten the generality of our results. §3 debates those suppositions and concludes that the rig used in those experiments is general to a large class of software problems.

1.1 Caveats

Before proceeding, we pause for a methodological digression. The conclusions of this paper are based on experiments with manipulating theories. The experimental basis for our conclusions is this paper's greatest strength. Opponents of our views can quickly refute our conclusions by designing and executing experiments that generate contrary conclusions. We would encourage such experiments. This paper is an attempt to add more rigor to software engineering research. If this paper inspires further careful experimentation, then it is a success even if those further experiments refute our conclusions.

On the other hand, the experimental basis for our conclusions is this paper's greatest weakness. Our conclusions stands only for the sample of problems studied here. If a different class of problems resulted in a different behaviour, the conclusion would have to be restricted to the class of problems seen in our above experiments. Further, all our conclusions fail if the restrictions we observe disappear due to some new method for theory manipulation. For examples, optimizations are always being discovered for theoretically intractable tasks.

Lest we prematurely convince you that this paper is fatally flawed, we hasten to add that none of our conclusions will based on runtimes. All our experiments will run to termination. That is,

the effects we report are not some function of processor speed or data structures. Instead, our conclusions come from giving our system unlimited time to perform its manipulations. Further, when we designed our experiments, we were nervous of our sampling bias. Hence, we always generated many, many variants of each sample problem. Nevertheless, our generators can not possibly generate problems across the entire space of problem types. At this time, we are unaware of such a class of problems and leave this issue for future research.

2 A Study into Theory Reusability

This section describes an example of the kinds of analysis that is possible via large-scale theory manipulation. We will describe an experimental rig and the behaviours seen in that rig. In summary, we will explore a suite of theories 378,000 times to conclude that the chances of building truly reusable components is a function of the amount of data used to test those components.

The intent of this section is to offer the reader a flavor of the kind of analysis we seek to encourage. Hence, for the moment, we will not discuss the external validity of these observations. In the sequel, however, we will argue that the observations seen is this rig are applicable to a wide class of software theories.

2.1 Motivation

Our reading of the literature is that design via reuse is the dominant paradigm in contemporary knowledge and software engineering. This paradigm dates back at least to 1964 with Alexander's work on architecture [Alexander 1964, Alexander, Ishikawa, Silverstein, Jacobsen, Fiksdahl-King & Angel 1977]. In summary, when we design via reuse, we re-shuffle components developed previously, then abstracted into a reusable form. Modern expressions of design via reuse include object-oriented design patterns [Buschmann, Meunier, Rohnert, Sommerlad & Stal 1996, Menzies 1997*a*], and the knowledge engineering research into ontologies [Gruber 1993] and problem solving methods [Schreiber, Wielinga, Akkermans, Velde & de Hoog 1994]. A tacit assumption in this paradigm is that a community shares a definition of some concept. This assumption would be violated if, in the usual case, different developers facing the same problems develop very different definitions of software concepts.

When won't reuse work? To answer that question, we assume that when we build reuse libraries, we are creating those libraries from an implicit space of alternative designs. In our experience, the design process stops when we have rejected an adequate number of clearly inferior proposals. When exploring the design space, there exists situations where software reuse is likely, and other situations where it is less likely. Suppose one good design was clearly superior to all alternatives. In this case, we could depend on different designers working at different times and

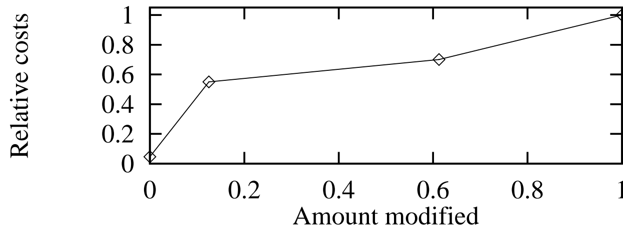

Figure 1: Relative of reuse with X% changes; i.e. cost of adaption divided by the cost of rebuilding from scratch. Data from the COCOMO-II software cost estimation project [Abts et al. 1998, p21].

different locations to arrive at the same final design. Reuse is likely in this case since the problem domain would force designers to make the same decisions. Designers would hence understand reusable artifacts since it would be clear why a particular reusable artifact was built *this way* and not *that way*.

Alternatively, suppose a clearly superior design was hard to detect. In this case, it is possible that different designers will arrive at different final systems. If the problem domain does not force a similar set of decisions, designers are less likely to understand supposedly reusable artifacts. Our designers may always be puzzling why a particular reusable artifact was built *this way* and not *that way*. Worse still, they may be tempted to tinker with the reusable artifact in order to contort it into their own view of the problem domain. Such tinkering can remove the economic justification for reuse. A learning curve must be traversed before any module can be adapted. By the time you know enough to change a little of that module, you may as well have re-written 60% of it from scratch; see Figure 1.

2.2 From Motivation to Manipulations

The above motivation can now be mapped into a series of theory manipulations. As mentioned above, during this mapping process, we will encounter certain restrictive **suppositions** that threaten the generality of our results. These suppositions will be noted, then debated later.

We begin by commenting that reuse is less likely if clearly superior designs are not recognizable. That is, some oracle cannot declare that theory T_1 is "much better than" theory T_2 (which we denote $T_1 \succeq T_2$). Let us assume that this oracle can access a list of output states that the program should reach from some inputs states. We define the *cover* of a theory T_i to be the largest number of outputs in can reach from the inputs. As $cover(T_i)$ approaches 100%, the theory can reach all its desired outputs. Using this definition of $cover(T_i)$, we can implement the oracle as follows:

$$T_i \succeq T_j \; if \; cover(T_i) \gg cover(T_j) \tag{1}$$

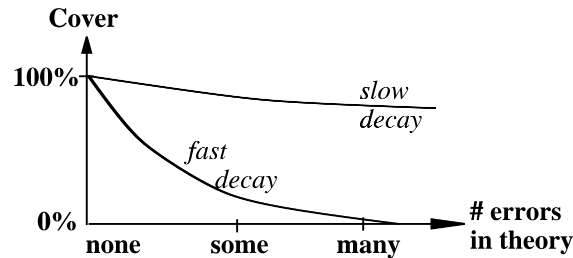

Figure 2: Decay curves for cover.

The first supposition of this experiment is:

Supposition 1 *Our oracle of better designs ($\succeq$) is widely applicable.*

Assuming that Equation 1 is valid, we note that as we introduce errors into our theories, we expect *cover* to decrease. Suppose the decay in *cover* was very rapid; e.g. as shown in the *fast decay* curve of Figure 2. In the case of fast decaying *cover*, we can hope that different designs will converge since, if ever we stumbled over a superior design, we would be sure to realize that it was indeed superior (since it's *cover* would be very much greater than its opposing theories). Alternatively, if the decay in *cover* was very shallow (e.g. the *slow decay curve* of Figure 2) then designs may not converge since competing theories would be indistinguishable.

The discussion in the above motivation section can now be expressed precisely using Equation 1 and the notion of *fast decay*:

> Reusable reuse libraries are likely in the case of fast decay since there will only be a small number (?one) of best designs on top of the decay curve.

Conversely:

> Reusable reuse libraries are unlikely in the case of slow decay since many designs exist on top of the decay curve. In this case, different designers working at different times and different locations may generate different solutions to the same problem.

Hence, to experimentally explore the likelihood of software reuse, we need an experimental rig that tracks the decay curves of a theory as we manipulate the theory to introduce errors.

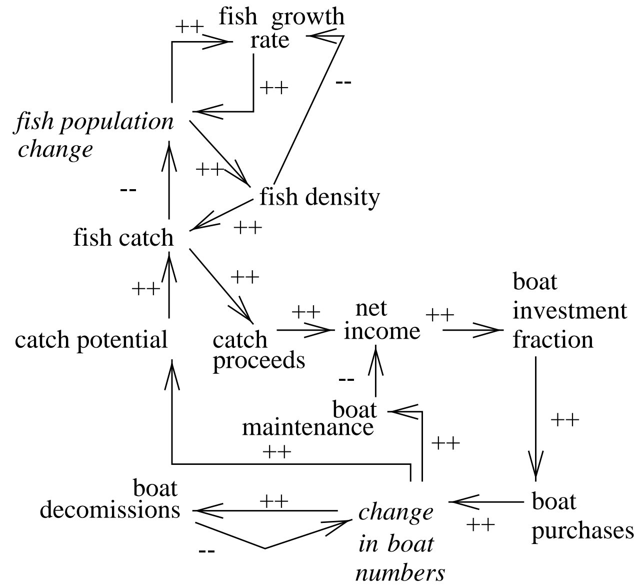

Figure 3: The fisheries model from [Bossel 1994] (pp135-141).

2.3 Experiment #1: Exploring Reuse

In this experiment, we generate *cover* decay curves of the form of Figure 2. To do this, we need (*A*) a sample theory to explore, (*B*) a method of introducing errors, (*C*) a method of generating input-output pairs that the correct theory should be able to cover, and (*D*) a simulator which can searching from inputs to find the maximum coverage of the output. This section describes our methods for implementing $< A, B, C, D >$.

2.3.1 *A*: Sample Theory

Figure 3 shows our sample theory. This figure is a qualitative form of some quantitative equations describing fishes growing in a fishery. For the moment, we do not discuss the generality of theories of the form of Figure 3. This issue will be discussed later.

In Figure 3, each theory variables has three states: *up*, *down* or *steady*. These values model the sign of the first derivative of these variables (i.e. the rate of change in each value). $X \stackrel{++}{\to} Y$ denotes that *Y* being *up* or *down* could be explained by *X* being *up* or *down* respectively. That is:

$$V_i \stackrel{++}{\to} V_j \;\; \equiv \;\; \left\{ \begin{array}{lcl} V_i \uparrow & \vdash & V_j \uparrow \\ V_i \downarrow & \vdash & V_j \downarrow \end{array} \right.$$

(where $\uparrow$ and $\downarrow$ denote up and down respectively.)

$X \overset{--}{\rightarrow} Y$ denotes that Y being *up* or *down* could be explained by X being *down* or *up* respectively. That is:

$$V_i \overset{--}{\rightarrow} V_j \equiv \begin{cases} V_i \uparrow & \vdash & V_j \downarrow \\ V_i \downarrow & \vdash & V_j \uparrow \end{cases}$$

Tacit in Figure 3 is conjunctions of influences. We should view this theory as influences splashing around pipes connecting tubs. Pairs of competing influences can cancel out. That is, we can explain the level of water in a tub remaining steady via conjunction of competing upstream influences; e.g.

$$((V_i \uparrow \vdash V_j \uparrow) \wedge (V_k \downarrow \vdash V_j \downarrow)) \vdash$$
$$((V_i \uparrow \wedge V_k \downarrow) \vdash (V_j = steady))$$

Figure 3 is a very small theory: 12 three-state variables copied 6 times (one for each time tick). Real world software systems can be much, much larger than 12 variables. Hence, our next supposition must be:

Supposition 2 *We can scale up conclusions from a small theory such as Figure 3 to larger theories.*

Apart from size, Figure 3 also has a fixed degree of connectivity, also known as fanout. The fanout of Figure 3 is $\frac{17\ edges}{12\ variables} = 1.4$. We will need to check that changing this fanout does not effect our conclusions. Hence:

Supposition 3 *We can scale conclusions from a theory of fanout=1.4 to theories with other fanouts.*

Apart from the size and fanout issues, experimenting with Figure 3 implies:

Supposition 4 *We can represent a useful range of software systems as networks like Figure 3.*

2.3.2 *B*: Introducing Errors

Figure 3 contains edges of two types: $X \overset{++}{\rightarrow} Y$ and $X \overset{--}{\rightarrow} Y$. We can corrupt the theory by randomly selecting M edges, then flipping their sign ($++$ to $--$ and visa versa). For a theory with N edges ($M \leq N$), we can now generate 2^M variants. Note that at $M = 0$ we have the original theory and, as M increases, we move to progressively worse theories.

2.3.3 *C*: Generating Input/Output Pairs

To generate input-output pairs that should generate 100% *cover*, we use the $M = 0$ theory (i.e. the correct theory) to generate data. To apply this method, the quantitative fisheries theory was run 15 times. Inputs and goals were generated by comparing all pairs of the measurements in the 15 runs (105 such pairs exist). That is, by comparing pairs of runs we can convert raw numbers to statements like (e.g.) "when we compare run 3 to 9, we saw that *catchPotential* went up and *catchProceeds* went down".

Next, the comparisons were converted into input-output pairs. The theory of Figure 3 was copied C times to represent the state of the system at time $C = 0, C = 1, \ldots$. Inputs are comparisons that appear at time copy $C = 0$ and outputs are comparisons that appear at time copy $C > 0$.

If we copy the theory C times, then we must link time copy $C = i$ to time copy $C = i+1$ using a *temporal linking policy*. Note the variables *Change in boatNumbers* and *fish population change*. These variables are the *explicit time variables*. In the XNODE temporal linking policy used in this first study, explicit time variables connects to themselves in the immediate future. That is, for each explicit time variable, XNODE adds a link

$$X_{C=i} \stackrel{++}{\rightarrow} X_{C=i+1}$$

We acknowledge that the choice of XNODE as the temporal linking policy is somewhat arbitrary and could limit the generality of our conclusions; i.e.

Supposition 5 *Conclusions drawn from a study of XNODE-based systems apply to non-XNODE systems.*

This test data library now contained measurements for all variables during the lifetime of the simulation (5 years; i.e. $C = 0, 1, 2, 3, 4, 5$). In practice, it is unlikely that all such data will be available to a testing team. Hence, we generate input-output pairs that refer to less than 100% of the variables in Figure 3. To generate test sets where *U* percent of the system was unmeasured, *U* percent of the goals was then discarded to produce 10 variants of the data with *U* at 0, 10, 20, 30, 40, 50, 60, 70, 80, and 90 percent unmeasured.

2.3.4 *D*: Simulating the Theory

A simulator for the fisheries model must be *contradiction-tolerant*. To see why, note that Figure 3 is a qualitative theory [Iwasaki 1989]; i.e. it is under-specified and hence can generate inconsistencies. For example, Figure 3 says that increasing *catchPotential* can increase *fishCatch*. It also says that decreasing *fishDensity* can decrease *fishCatch*. It is undetermined what happens to *fishCatch* in the case of increasing *catchPotential* AND decreasing *fishCatch*. Since Figure 3 does

not specify the relative sizes of these influences on *fishCatch*, all we can say is that *fishCatch* can rise, fall, or remain steady (in the case where the two competing influences cancel each other out). Hence, if a simulator arrives at *fishCatch* from increasing *catchPotential* and decreasing *fishCatch*, then it must fork the reasoning one way for each possibility; i.e. once for *fishCatch=up*, once for *fishCatch=down*, and once for *fishCatch=steady*.

In this experiment, our contradiction-tolerant simulator will be the HT4 abductive inference engine [Menzies & Waugh 1998, Menzies & Michael 1999, Menzies 1995, Menzies 1996*a*, Menzies 1996*b*, Menzies & Compton 1997]. Abduction is informally defined as inference to the best explanation (e.g. [O'Rourke 1990]). More formally, abduction is the search for assumptions which, when combined with some theory explains some set of goals [Eshghi 1993]. Mutually exclusive assumptions imply that many alternative abductive-explanations may be generated. Such mutually exclusive assumptions must be managed in seperate *worlds* of beliefs. Returning to the above example, in the case of increasing $catchPotential$ and decreasing $fishCatch$, then we generate three worlds W_1, W_2, W_3. W_1 includes $fishCatch = up$, W_2 includes $fishCatch = down$, and W_3 includes $fishCatch = steady$.

In essence, HT4 is like a search engine exploring some network of ideas looking for useful and consistent portions (each such portion is a world). The generality of our conclusions depends on this style of search being applicable to a range of software engineering problems; i.e.

Supposition 6 *We can model software construction as the exploration of networks like Figure 3.*

Supposition 7 *We can model network exploration as abduction.*

One trap with using HT4 is that our conclusions may only hold when processing indeterminate theories. That is:

Supposition 8 *Our conclusions from the study of indeterminate theories applies to determinant theories.*

2.3.5 Details

Summarizing the above discussion, our experimental rig contains a theory T generated via some time linking policy (e.g. XNODE). The theory is a directed graph containing vertices and edges $< V, E >$. Each theory vertex V is a pair of a variable and a time copy index C_i ; e.g. $fishCatch@C_2$ is the value of $fishCatch$ in time copy $C = 2$. E are the connectors between variables are one of a set of pre-defined types; e.g. $\stackrel{++}{\rightarrow}$ or $\stackrel{--}{\rightarrow}$. That is:

$$
\begin{aligned}
V_i &= variable@time\,copy\,index \\
E_i &= V_i \stackrel{++}{\rightarrow} V_j \;or\; V_i \stackrel{--}{\rightarrow} V_j \\
T &= < V, E >
\end{aligned}
$$

Our goals we are trying to explain are the outputs. In the general case, only some of the outputs are reachable. The reachable outputs are denoted $output'$. Recalling the above discussion on XNODE, all the outputs come from time copy $C > 0$. That is:

$$outputs' \subseteq outputs \subseteq V@C_{i>0}$$

The assumptions are those variables used to connect used inputs to used outputs. The used inputs are denoted $inputs'$. Recalling the above discussion on XNODE, all the inputs come from time copy $C = 0$. That is:

$$\begin{aligned} inputs' &\subseteq inputs \subseteq V@C_0 \\ A &\subseteq V - inputs' - outputs' \end{aligned}$$

U is the percentage of the theory variables that do not appear in the inputs and outputs. That is:

$$U = \left(1 - \frac{|inputs \cup outputs|}{|V|}\right) * 100$$

An *abductive-explanation* is a *world* of mutually compatible beliefs W_i. A world connects some inputs to some outputs without causing contradictions. In HT4, worlds are extracted from the edges in our theories. That is:

$$W_i \subseteq E \quad (2)$$

$$W_i \cup inputs' \cup A \vdash outputs' \quad (3)$$

$$W_i \cup inputs' \cup A \nvdash \perp \quad (4)$$

HT4 seeks the abductive explanation W_i with maximal *cover*; i.e. the explanation that maximizes the size of $output'$. That is:

$$cover(T) = max(|W_i \cup outputs'|) \quad (5)$$

With these definitions in hand, we can now specify our experimental rig. 20 times we flippped the edge type of between none to all of the edges in Figure 3 to create a corrupted theory T_x. For each such corrupted theory T_x, we:

- Copied T_x 6 times (once each for $C = 0, 1, 2, 3, 4, 5$).
- Connected the copies using the XNODE linking policy to form T_y.
- For all 105 comparisons of the data, we:

- For $U \in \{0, 10, 20...90\}$, we:
  - Built the outputs and inputs using $100 - U\%$ of the comparison data.
  - Called HT4 to find maximum cover using the inputs, the outputs and T_y.

There are 17 edges in Figure 3. Hence, this procedure calls HT4 378,000 times (20 repeats * 0..17 edges flipped * 105 comparisons * 10 U values).

2.3.6 Results

The behaviour of this experimental rig is shown in Figure 4. To read the results, note that the x-axis shows a progression from a correct fishery theory (at 0 edges flipped) to a very incorrect fisheries theory (at 17 edges flipped). For the top plot, the y-axis shows the maximum cover found by HT4. As we would expect, as the number of errors in a theory increases, the coverage of desired goals decreases. Recalling our above discussion, reuse is unlikely in the case of slow decay of the cover curve. Our reading of Figure 4 is that after $U = 70$, the decay curve becomes very flat; i.e. it becomes difficult to distinguish a theory with no errors (left hand side of the top plot in Figure 4) from a theory with multiple errors (right hand side of the top plot in Figure 4).

The runtime results (Figure 4, bottom) show two interesting trends. Firstly, in nearly all cases, as the theories grow more corrupted, it becomes faster to determine what goals are explicable (exception: for the very unmeasured theories, the runtimes for good theories the same as the runtimes of bad theories). This is a nice result: nonsense theories can be rejected faster than sensible theories.

Secondly, the above experiment tried to find an explanation for every member of the outputs. That is, as the percentage of unmeasured variables U was increased, the size of the output goal set decreased. A pre-experimental intuition was that as we tried to explain more and more goals, the runtimes would increase. For theories that were mostly correct (corruptions less than 3 out of 17), this intuition proved correct. However, for theories that are moderately to wildly incorrect (more than 3 of the 17 edges corrupted), the runtimes decreased as the number of goals was increased. We explain this surprising result as follows. For theories that are mostly correct, many successful proofs of goals can be generated. The overheads of building these proofs can slow down the inference procedure. On the other hand, as we supply more and more data for moderately to wildly incorrect theories, we constrain the search space more and more. For example, in fisheries, an unmeasured variable could take the value *up*, *down*, or *steady*. Once that variable is measured, two-thirds of the possible search space disappears. Hence, for these not-very correct theories, runtimes decrease as we increase the number of goals.

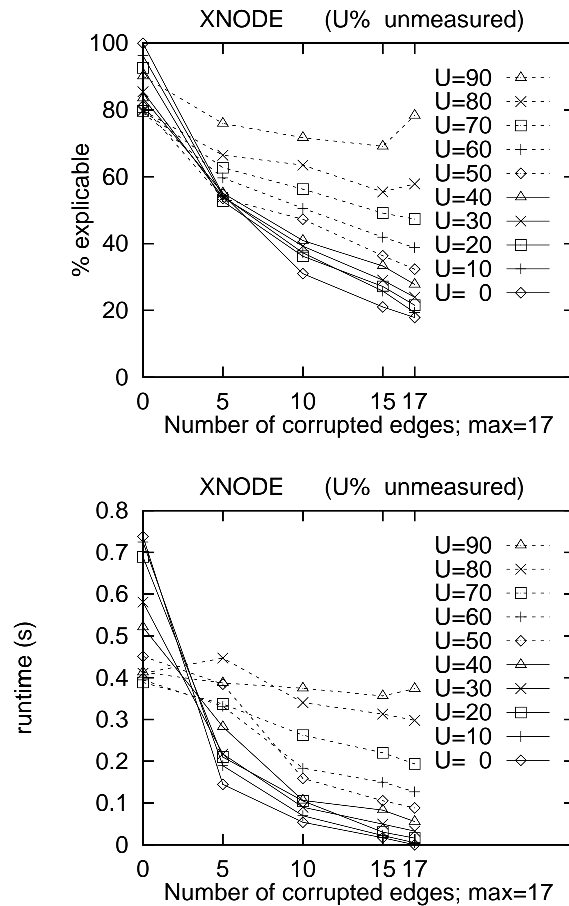

*Figure 4: Experiment #1. XNODE linking: Corrupting 0 to 17 edges, running validation with less and less data. Percentage explicable (*cover*) on top; runtimes on bottom.*

2.3.7 Experiment #1: Conclusions

At least for the experimental rig shown above, the likelihood of reusable components is a function of the amount of data available to test the components. As less and less of the theory is unmeasured in the inputs/outputs set, the harder it becomes to recognize a clearly superior theory. The threshold appears to be around 70% unmeasured theory variables.

If this is a general result, then we must doubt the utility of building reuse libraries. For nearly a decade we have been researching software testing. In our experience, we believe that it is very rare that software test suites refer to more than 30% of the variables in a component. If we test our components with insufficient data, then we will not detect clearly superior designs. In this case,

designers working in different locations may arrive at different final designs for their software components. That is, we expect that:

- Supposedly reusable software components contain design decisions that are only acceptable to the community that made them.
- A broader audience might find those decisions confusing or unacceptable.
- Hence, software components would not be widely reused.

There are at least two cases in which we could demonstrate that this conclusion is not general:

1. We could show that our suppositions are overly restrictive.
2. We could show that reuse is a common and successful technique.

We will discuss the suppositions in the next section. As to the second point, it has yet to be conclusively proved that reuse is a common and successful technique. Menzies [Menzies 1998] criticizes overly-enthusiastic reports of reuse that lack some measure of quality of the constructed system. Without such quality assessments, it is hard to assess the overall impact of reuse[1]. Proponents of reuse rarely track the on-going costs of maintaining with those components [Menzies, Cukic, Singh & Powell 2000] (exception: [Lim 1994]). Most reuse reports do not clearly distinguish between *verbatim reuse* and *reuse with some tinkering*. Recalling Figure 1, such tinkering to customize a reusable component can significantly increase the cost and decrease the benefits of using reusable components. Further, even enthusiastic proponents of reuse testify to the lack of widespread reuse. For example:

> I do not think we have yet succeeded in software reuse. In the State of the Practice, I do not see a lot being applied. Yes, OO class frameworks, Java, Visual Basic, etc. have facilitated the code reuse, but less is done, in general, at the higher level such as in the design and requirements phases of a project. In the State of the Art, I have not perceived any incremental progress although many issues have been addressed [Guerrieri 1999].
>
> Reuse continues to be a problem whose potential remains elusive. Each new solution remains full of promise but riddled with what look like insurmountable problems. [Perry 1999]

Even reuse enthusiasts such as Frakes [Frakes 1999] caution that there exist significant practical problems with the widespread proliferation of reuse libraries; e.g. problems with searching the reuse library.

In summary, our reading of the literature is that we can't reject the generality of the above conclusions based on successful reports of reuse.

[1]One of the few reuse reprots that includes quality measures is [Lim 1994]. However, that report refer to intra-institutional reuse, not widespread inter-institutional reuse.

3 External Validity

We believe that the above results are general to a wide class of software. The rest of this article attempts to justify that that belief. To make this case, we explore the suppositions described above. In the process of making that case, we will propose and perform other experiments that comment positively on the practicality of processing indeterminate theories (including early life cycle requirements theories).

3.1 Choice of Oracle

Supposition 1 claimed that theories should be assessed via how well they can reproduce a set of desired goals (recall Equation 1). Clearly, there are many other ways to assess a theory; e.g. parsimony, runtime speed of its inferences, etc. However, we regard these other measures as secondary. Theories only become interesting when they can perform their primary tasks. Only after demonstrating theory competency should we move on to assess (e.g.) its speed or economy of expression. Partial support for our view on theory assessment can be found elsewhere in the literature. For example, the model checking community assess theories via their coverage of temporal logic constraints; e.g. [Holzmann 1997, Clarke, Emerson & Sistla 1986].

3.2 Bigger Theories

Supposition 2 claimed that a conclusion drawn from a theory with 12 variables can be extrapolated to larger theories. Recall that our key observation was that our test oracle failed to distinguish good theories from bad theories after $U = 70\%$. What would happen to this observation if the theory gets larger?

In answering this question, we first note that testing gets harder as the system gets larger. Some support for this claim can be found in NASA's experience with model checkers. The informal rule-of-thumb at NASA is that model checking techniques such as SPIN [Schneider, Easterbrook, Callahan, Holzmann, Reinholtz, Ko & Shahabuddin 1998, Holzmann 1997] can't process large systems. The invariants from merely 30 classes in an OO system can require one gigabyte of memory to model check (and one gigabyte of ram is our current practical limit of what we can expect on a workstation).

Note that if we can't tell good from bad even for very small theories then, as the theory gets larger than Figure 3, it must become even harder to distinguish good larger theories from bad larger theories. Hence, we believe that Supposition 2 holds since our pessimistic conclusions concerning reuse must get even more pessimistic as system size grows.

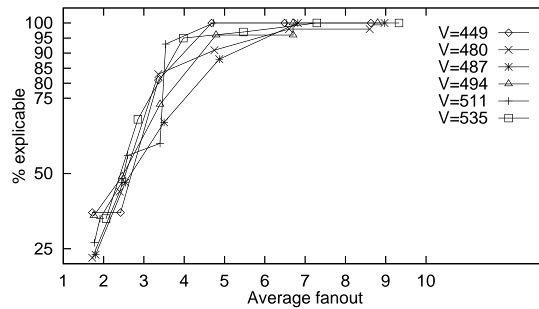

Figure 5: Percentage explainable. From [Menzies 1996b].

3.3 More Intricate Theories

Supposition 3 was a variant of Supposition 2. In this supposition, we claimed that conclusions from a theory with fixed fanout (1.7) applies to theories with different fanouts. In earlier work, Menzies studied how different fanouts effect the conclusions of HT4 [Menzies 1996*b*]. In that experiment, edges were added at random to variants of a real-world theory of human glucose regulation taken from [Smythe 1989]. This theory was written in the same format as Figure 3 and is described in detail in [Menzies & Compton 1997]. That theory had 1246 edges and 554 variables (fanout=1246/554=2.25). Six "clones" of that theory were generated using random variables whose distributions were taken from the glucose theory (e.g. the mean number of parents in the clones was the same as the mean number of parents in the glucose theory). Each clone was approximately the same size as the glucose theory and contained 449, 480, 487, 494, 511, or 535 nodes. These were executed using HT4 with randomly chosen inputs and goals. Edges were then added randomly and the executions repeated. Figure 5 shows the results. At low fanouts, many behaviours were inexplicable. However, after a fanout of 4.4, most behaviours were explicable. Further, after a fanout of 6.8, nearly all the behaviours were explicable.

In summary, Menzies found that as fanout increases, the probability of explaining any randomly selected output also increases. As this probability increases, the *cover* decay curve would flatten out since our ability to distinguish a theory with N errors from a theory with $N + 1$ errors would decrease. Consequently, we argue that Supposition 3 holds: our pessimism about detecting a superior theory would *increase* if the fanout increases from the value of 1.7 seen in Figure 3.

3.4 Software = Network of Connections

Supposition 4 claimed that a wide range of software can be characterized as a network of connections between variables. This is hardly a controversial position. Many researchers model software as a network of connecting influences:

- Software coverage tools often view software as a network. Good test suites try to maximize the coverage of network components (e.g. [Bieman & Schultz 1992, Harrold, Jones & Rothermel 1998, Weyuker 1993]).
- Any optimizing compiler builds a network (control/data flow graph) from the code. Optimization is then a matter of reorganizing the network to speed up the program.
- Model checkers access such a network when they build a space of program variable states.
- Expert systems and logic programs can be reduced to such structures (using a technique called *partial evaluation* [Sahlin 1991]).

3.5 Beyond XNODE

Our experiment assumed Supposition 5, i.e. for all explicit time variables X, we connect X at time copy index C_i to X at time copy index C_{i+1}. What happens to our conclusions if we vary the time linking policy? This question is explored in experiment #2.

3.5.1 Experiment #2: Other Temporal Linking Policies

Experiment #2 repeats the experimental rig of Experiment #1, but with a different temporal linking policy. In the XEDGE (explicit edge) linking policy, time changes on the out-edges from the explicit time variables. That is, for each variable Y downstream from an explicit time variable X, XEDGE adds a link $X_{T=i} \rightarrow Y_{T=i+1}$. For example, recall from Figure 3 that *fishGrowthRate* is connected to *fish population change* as follows:

$$fishPopulationChange \stackrel{--}{\rightarrow} fishGrowthRate$$

XEDGE would add the edge:

$$fishPopulationChange_{C=i} \stackrel{--}{\rightarrow} fishGrowthRate_{C=i+1}$$

Figure 6 shows the behaviour seen in Experiment #2. The curves have numerous similarities to the XNODE behaviour seen in Figure 4:

- As before, nonsense theories can be rejected faster than sensible theories since the runtimes decrease as add errors.
- As before, there exists a lower threshold in test suite size, below which we cannot distinguish good theories from bad theories. That is, once again, we see that as we offer less and less data in the test suite, the *cover* decay curves flatten out.

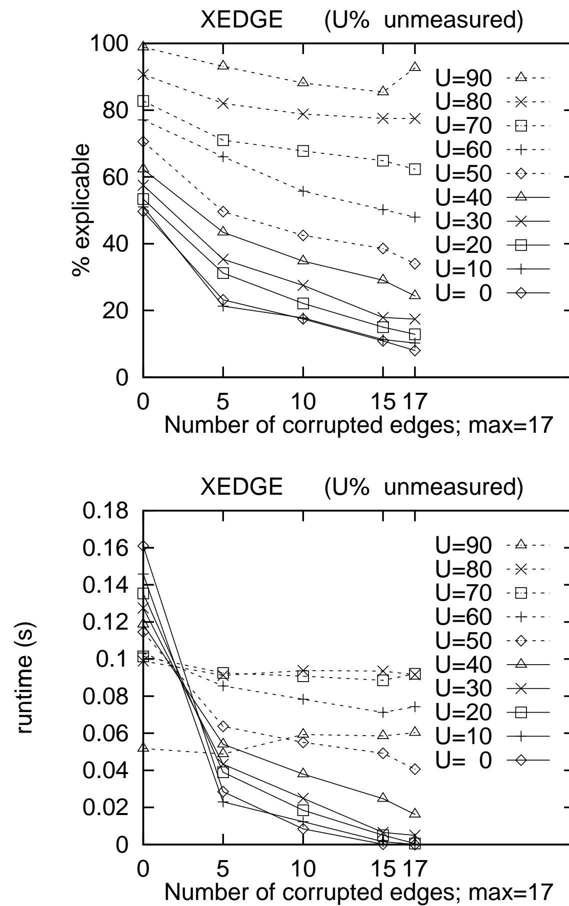

Figure 6: Experiment #2. XEDGE linking: Corrupting 0 to 17 edges, running validation with less and less data. Percentage explicable on top; runtimes below.

Elsewhere we have explored six other temporal linking policies [Waugh, Menzies & Goss 1997]. The above two features are noted in all those experiments. That is, Supposition 5 holds for at least seven other temporal linking policies than XNODE.

3.6 Software Construction = Network Exploration

Supposition 6 argues that when we search Figure 3, we are doing something that is important to a large class of software systems. We take this view from Herbert Simon. Simon describes design as the "quintessential human activity" [Simon 1969]. Everything we do as agents in the world requires

design solutions to the problems that face us. Hence, to Simon, studying the design problem was a method of studying the general problem of artificial intelligence (AI).

Simon's research characterized design and artificial intelligence as a search problem. In this framework, design is viewed as the traversal of a space of possibilities, looking for pathways to goals. This space can be explicit or implicit:

Explicit: e.g. taken from a library of software components.

Implicit: e.g. extracted from the designer's mind as they imagine different possible, but as yet unbuilt, software systems.

Modern expressions of this approach of design-as-search includes most of the AI search literature [Pearl & Korf 1987] and the SOAR *knowledge-level* (KL) search [Newell 1982, Newell 1993]. In a KL search, intelligence is modeled as a search for appropriate operators that convert some current state to a goal state. Domain-specific knowledge is used to select the operators according to *the principle of rationality*; i.e. an intelligent agent will select an operator which its knowledge tells it will lead the achievement of some of its goals.

This view as design-as-search through a network is not just an AI concept. OO designers perform such a search as they search through their class libraries looking for reusable code. More generally, any time a software designer pauses to consider *this option* vs *that option*, they are mentally exploring a networks of options.

3.7 Network Exploration = Abduction

Supposition 7 noted that we used abduction as our network exploration tool. We could reject our conclusions if it could be shown that abductive inference is relevant only for a very small class of software tasks.

Our view of abduction can can be summarized as follows: seek islands of consistency within a space of possibly contradictory ideas. More precisely, make what inferences you can that are relevant to some goal (Equation 3), without causing any contradictions (Equation 4). When more than one solution exists to Equation 3 and Equation 4, then use some BEST operator to select a preferred set of solutions. Clearly, this procedure is more general than just a description of "inference to the best explanation". Many tasks have an abductive mapping, as shown in Figure 7. Hence, with one caveat, we argue that the use of an abductive inference engine *increases* the generality of our conclusions. The one caveat is as follows. Abduction is an indeterminate inference procedure. Standard software strives to be determinant. The issue of indeterminacy is discussed below.

3.8 Implications of Indeterminacy

The above discussion argued that Suppositions 1 through 7 did not overly restrict the generality of our conclusions. Only Supposition 8 remains: that conclusions drawn from HT4's indeterminate inference are generally applicable. Reviewing the tasks implemented by abduction (see Figure 7), it is clear that most of those tasks come from the knowledge engineering community. Knowledge-based systems often use heuristics to simplify complex tasks. Heuristics can introduce indeterminacy into a theory. Heuristics are just guesses and different guesses can drive program to very different conclusions.

Supposition 8 threatens the generality of our conclusions. HT4 is a multiple world reasoner and conventional software is determinant. That is, software constructed from standard software engineering techniques only ever generates a single assignments to variables. Conclusions drawn from HT4 will not apply to standard software systems if its multiple-world reasoning confuses the task of testing software.

To explore this issue, we need more data. The average number of worlds generated by HT4 in Experiment #1 is shown in Figure 8. Note the low total average number of worlds (max=1.28). While HT4 *can* support multiple worlds, these experiments generated very few. That is, the conclusions made above regarding reuse were not made in a wildly indeterminate device. This absence of wild indeterminacy persists, even if we make major changes to the HT4-rig. For example, fisheries is a sparsely connected theory: 12 variables connected by 17 edges (fanout=17/12=1.4). Perhaps theories with higher fanouts generated more explanations? This hypothesis was tested in **Experiment # 3**. That experiment added in 0, 5, 10, 15, 20, 25 or 30 new edges at random, checking all the time that the added edges did not exist already in the theory. This generated theories with fanouts ranging from 1.4 to (17+30/12=3.9). The theory with the new edges was then copied over 5 time steps and connected via XNODE and executed in the same manner as experiment #1. The results are shown in Figure 9. Note that the number of worlds generated is not much more than before (1.58 maximum). Hence, we could not blame the low number of worlds seen in HT4 output on restrictive fanouts.

- Abductive-validation asks the question "What is the maximum percentage of known/desired behaviour that can be explained by some theory?". This can be implemented as Equation 5; i.e. use a BEST that favors explanations with the largest intersection to the output goal set [Menzies 1996*b*, Menzies 1995, Menzies & Compton 1997]
- Minimal-fault diagnosis means using a BEST that favors explanations with the fewest possible inputs and the most known goals [Reggia, Nau & Wang 1983] (this is only one of a range of abductive approaches to diagnosis: other approaches are described in [Console & Torasso 1991]).
- Abductive-classification uses a BEST that favors explanations with the largest number of most-specific concepts [Poole 1985].
- Abductive-explanation uses a BEST that favors explanations with the largest number of concepts that are familiar to the user [Leake 1993].
- Abductive-planning uses a BEST that favors explanations with the most goals and the least total cost.
- Abductive-monitoring means caching the explanations generated by abductive-planning, then deleting any explanations whose assumptions contradict any new data.
- Abductive-prediction uses a BEST that favors explanations with the most number of future events that interest you.
- Abductive requirements engineering is discussed in [Zowghi, Ghose & Peppas 1996, Menzies, Easterbrook, Nuseibeh & Waugh 1999]. Suppose we know who wrote each portion of a theory. Conflicts between users can be detected when different people's ideas end up in different explanations. These conflicts can then be negotiated by focusing on the key conflicting guesses that split the explanations. Disputes been feuding users can be adjudicated as follows: the winning user offers theories which build explanations that can explain most of the desired behaviour.
- Abduction is also applicable to numerous other areas including legal reasoning [Kowalski & Toni 1996]; visual pattern recognition [Poole 1990]; financial reasoning [Hamscher 1990]; diagrammatic reasoning [Menzies & Compton 1994]; machine learning [Hirata 1994]; case-based reasoning [Leake 1993]; natural-language processing [Ng & Mooney 1990]; and analogical reasoning [Falkenhainer 1990]. For more on the applications of abduction, see [Menzies 1996*a*, Kakas, Kowalski & Toni 1998].

*Figure 7: Tasks with an abductive mapping. Adapted from [Menzies 1996*a*].*

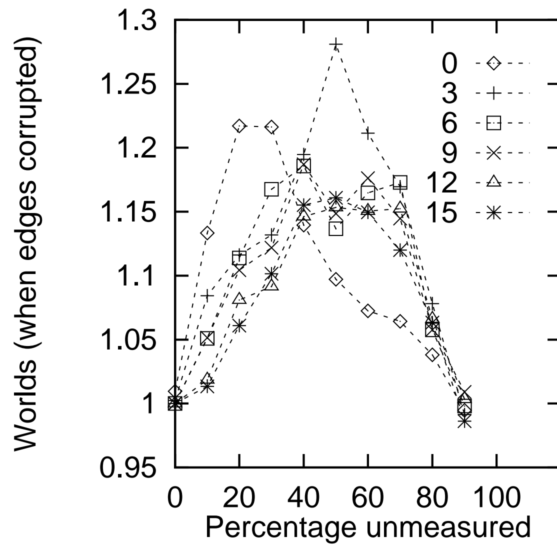

Figure 8: Experiment #1: Average worlds seen as we corrupt 0,3,.. 15 edges from Figure 3 and change the percentage of theory variables unmeasured in the test set (changing $U\%$).

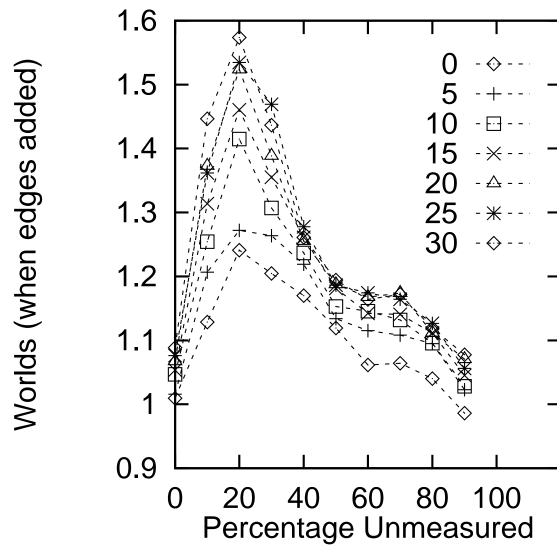

Figure 9: Experiment #3. Worlds generated edges with XNODE after adding 0,5,10,15,20,25,30 edges.

Nor can we blame the missing worlds on the temporal linking policies used in these experiments. Recall that of the 12 variables in the fisheries theory, only 2 of them made cross-time connections. Perhaps more worlds would be observed if more connections were made across time. **Experiment #4** tested this hypothesis. A new temporal linking policy was defined. In the INODE (implicit node) linking policy, any variable X at time copy C can effect that variable at time copy $C + 1$; i.e. for every variable we add the rule:

$$X_{C=i} \stackrel{++}{\rightarrow} X_{C=i+1}$$

Experiment #4 repeats experiments #1 and #3 using INODE instead of XNODE. The results are shown in Figure 10. The extra time links did lead to more explanations, but not many more (maximum=5).

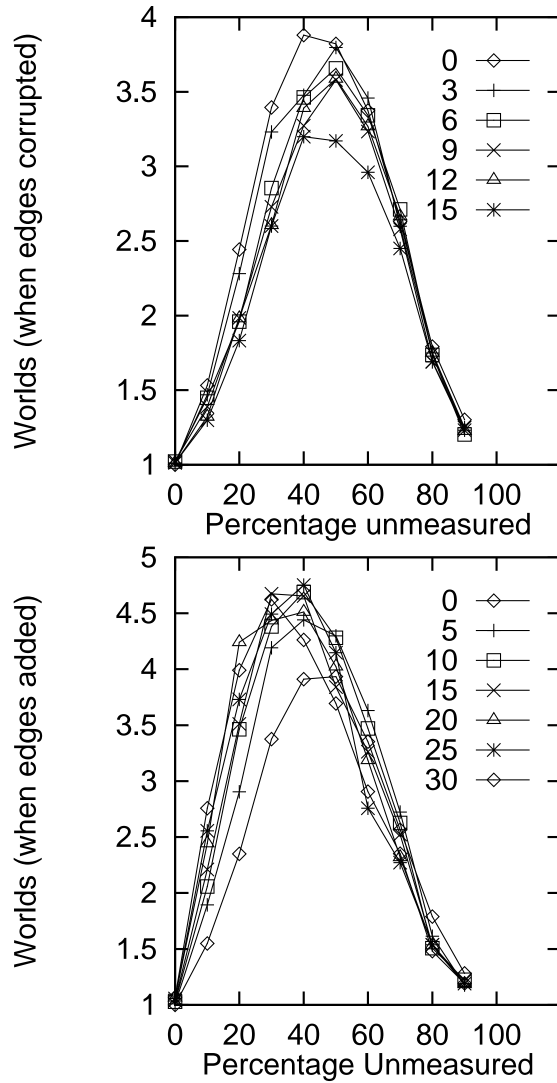

Figure 10: Experiment #4. Effects of using INODE. Top: flipping the annotation on 0,3,6,9,12,15 edges. Bottom: adding 0,5,10,15,20,25,30 edges.

Since HT4 and HT4-like systems do not generate wild indeterminacy, we can endorse supposition 8. Still, we must explain these repeated failures to generate wild indeterminacy. We have two hypotheses:

- [Menzies 1996*a*] noted that HT4 uses *set-covering abduction*; i.e. it builds worlds only from variables that connect inputs to goals. An alternative approach is consistency-based abduction used in (e.g.) the ATMS [DeKleer 1986] which builds worlds from all variables in the theory (for more on these two types of abduction, see [Console & Torasso 1991]). Set-covering can use fewer variables than consistency-based approaches. Hence, set-covering systems like HT4 may generate fewer worlds than expected.

- Suppose our software contains a small number of *master variables* that control the larger number of slave variables in the rest of the system. In such master-variable systems, worlds need only be generated only for conflicts in the master variables; i.e. conflicts in the larger number of slave variables do not matter. Note that if there are very few master variables, then we need only generate very few worlds. Further, we can find those worlds very easily. Any randomly generated pathway from goals back to inputs have a high chance of using the master variables. After generating a small number of such random paths, an inference procedure would find the key assumptions within the master variables. Elsewhere, with Cukic and Singh, Menzies has offered evidence such master-variable systems are common. This evidence comes from theory [Menzies et al. 2000], experimentation [Menzies et al. 1999], and an extensive literature review [Menzies & Cukic 2000].

The absence of wild indeterminacy in HT4-like systems has significant implications for the construction of environments that support debates between (e.g.) multiple-stakeholder during requirements engineering. Requirement engineering researchers such as Easterbrook [Easterbrook 1991], Finkelstein [Finkelstein, Gabbay, Hunter, Kramer & Nuseibeh 1994], and Nuseibeh [Nuseibeh 1997] argue that we should routinely expect specifications to reflect different and inconsistent viewpoints. A key process within deliberation and negotiation is supporting arguments. Argumentation support environments can be swamped by a surplus of choices. 20 yes-no choices means $2^{20} = 1,000,000$ design choices. Elaborate mechanisms have been built for implementing such arguments; e.g. [Menzies 1997*b*, Mylopoulos, Chung & Nixon 1992, Nuseibeh & Russo 1999, Rich & Feldman 1992, Boehm, Egyed, Port, Shah, Kwan & Madachy 1998]. The results of our experiments suggest that these mechanisms may be overly elaborate; e.g. when we search across that space of 2^{20} options from goals to inputs, we find that our search paths need only a few worlds. HT0 is a very simple variant of HT4 that exploits this "a few worlds are enough" property [Menzies & Michael 1999]. HT0 assumes master-variable systems. The algorithm generates a small number of randomly selected worlds in sequence; i.e. W_0, then W_1, then W_2 and so on. HT0 terminates when

the cover of W_i is not much greater than the maximum cover found by worlds $W_0, W_1 \ldots W_{i-1}$. etc.. Experiments with HT0 show that:

- Maximum cover plateaus very quickly. That is, HT0's partial world search can terminate much faster than HT4's rigorous exploration of all worlds. Hence, HT0 is much faster than HT4. HT4 explores all worlds. In experiments with one implementation of that algorithm, HT4 was observed to run in exponential time (HT4 crashed when running theories bigger than 1,000 horn clauses). HT0, running on the same platform, executed in polynomial time and has terminated on very large theories (up to 18,000 horn clauses).

- For problems where HT4 terminates (i.e. theories with < 1000 horn clauses), HT0 finds 98% of the maximum cover found by HT4. That is, HT0's partial world search covered nearly as much of the theory at HT4's rigorous exploration of all worlds.

The above experiments, plus the HT0 experience, suggest that exploring conflicts within requirements is easier than we might have thought. This possibility is explored extensively elsewhere [Menzies et al. 1999].

4 Conclusion

Based on four experiments that performed 100,00s of manipulations on a theory, we made two counter-intuitive findings:

- Reuse is harder then we had thought. Our supposedly reusable components will not converge to similar designs unless we can explore those components with test suites that are larger than those seen in current practice.

- Exploring the space of contradictory ideas seen in (e.g.) requirements engineering is easier than we had thought. A small number of random probes across the space of possible conflicts will find most of the worlds.

We conjecture that these results are counter-intuitive since our intuitions were based on experience with dozens, not 100,000s of systems. Our expectations of software development is based on our experience. Over the course of a single professional's career, an individual may see several dozen systems (at most). From those few dozen examples, that individual forms perceptions of the core issues and the general rules of software engineering. In this paper we have explored methods of finding some general rules of software engineering based not on dozens, but 100,00s of manipulations of a theory.

A side-effect of the above analysis is the evolution of a general method for exploring issues within software development:

1. Collect representative theories.
2. Define representative manipulations.
3. Execute many manipulations over the many theories.
4. Summarize the results.
5. Consider the implications of those summarizes on standard practice.

We think that this method is a powerful technique. For example, our conclusions can be demonstrated incorrect if, after applying these five steps, other researchers generate very different summaries to those described above.

Acknowledgements

This paper has benefited enormously from the comments of several anonymous referees. Simon Goss provided invaluable support in developing this material. Aditya Ghose and Steve Easterbrook offered insightful comments into earlier drafts of this paper. During this research, the first author was supported by the AI Department University of NSW, and then a NASA cooperative agreement (#NCC 2-979).

References

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

Alexander, C. [1964], *Notes on Synthesis of Form*, Harvard University Press.

Alexander, C., Ishikawa, S., Silverstein, S., Jacobsen, I., Fiksdahl-King, I. & Angel, S. [1977], *A Pattern Language*, Oxford University Press.

Bieman, J. & Schultz, J. [1992], 'An empirical evaluation (and specification) of the all-du-paths testing criterion', *Software Engineering Journal* **7**(1), 43–51.

Boehm, B., Egyed, A., Port, D., Shah, A., Kwan, J. & Madachy, R. [1998], 'A stakeholder win-win approach to software engineering education', *Annals of Software Engineering* **6**, 295–321.

Bossel, H. [1994], *Modeling and Simulations*, A.K. Peters Ltd. ISBN 1-56881-033-4.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. & Stal, M. [1996], *A System of Patterns: Pattern-Oriented Software Architecture*, John Wiley & Sons.

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

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

Console, L. & Torasso, P. [1991], 'A Spectrum of Definitions of Model-Based Diagnosis', *Computational Intelligence* **7**, 133–141.

DeKleer, J. [1986], 'An Assumption-Based TMS', *Artificial Intelligence* **28**, 163–196.

Easterbrook, S. [1991], Elicitation of Requirements from Multiple Perspectives, PhD thesis, Imperial College of Science Technology and Medicine, University of London. Available from `http://research.ivv.nasa.gov/~steve/papers/index.html`.

Eshghi, K. [1993], A Tractable Class of Abductive Problems, *in* 'IJCAI '93', Vol. 1, pp. 3–8.

Falkenhainer, B. [1990], Abduction as Similarity-Driven Explanation, *in* P. O'Rourke, ed., 'Working Notes of the 1990 Spring Symposium on Automated Abduction', pp. 135–139.

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

Frakes, W. [1999], Domain engineering education, *in* 'Proceedings of WISR9: The 9th annual workshop on Institutionalizing Software Reuse'. Available from `http://www/umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Frakes.html`.

Gruber, T. [1993], 'A translation approach to portable ontology specifications', *Knowledge Acquisition* **5**(2), 199–220.

Guerrieri, E. [1999], Reuse success - when and how?, *in* 'Proceedings of WISR9: The 9th annual workshop on Institutionalizing Software Reuse'. Available from `http://www/umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Guerrieri.html`.

Hamscher, W. [1990], Explaining Unexpected Financial Results, *in* P. O'Rourke, ed., 'AAAI Spring Symposium on Automated Abduction', pp. 96–100.

Harrold, M., Jones, J. & Rothermel, G. [1998], 'Empirical studies of control dependence graph size for c programs', *Empirical Software Engineering* **3**, 203–211.

Hirata, K. [1994], A Classification of Abduction: Abduction for Logic Programming, *in* 'Proceedings of the Fourteenth International Machine Learning Workshop, ML-14', p. 16. Also in *Machine Intelligence* 14 (to appear).

Holzmann, G. [1997], 'The model checker SPIN', *IEEE Transactions on Software Engineering* **23**(5), 279–295.

Iwasaki, Y. [1989], Qualitative physics, *in* P. C. A. Barr & E. Feigenbaum, eds, 'The Handbook of Artificial Intelligence', Vol. 4, Addison Wesley, pp. 323–413.

Kakas, A., Kowalski, R. & Toni, F. [1998], The role of abduction in logic programming, *in* C. H. D.M. Gabbay & J. Robinson, eds, 'Handbook of Logic in Artificial Intelligence and Logic Programming 5', Oxford University Press, pp. 235–324.

Kowalski, R. & Toni, F. [1996], 'Abstract argumentation', *Artificial Intelligence and Law Journal* **4**(3-4), 275–296.

Leake, D. [1993], Focusing Construction and Selection of Abductive Hypotheses, *in* 'IJCAI '93', pp. 24–29.

Lim, W. [1994], 'Effects of reuse on quality, productivity, and economics', *IEEE Software* pp. 23–30.

Menzies, T. [1995], Principles for Generalised Testing of Knowledge Bases, PhD thesis, University of New South Wales. Avaliable from http://www.cse.unsw.edu.au/~timm/pub/docs/95thesis.ps.gz.

Menzies, T. [1996*a*], 'Applications of abduction: Knowledge level modeling', *International Journal of Human Computer Studies* **45**, 305–355. Available from http://www.cse.unsw.edu.au/~timm/pub/docs/96abkl1.ps.gz.

Menzies, T. [1996*b*], On the practicality of abductive validation, *in* 'ECAI '96'. Available from http://www.cse.unsw.edu.au/~timm/pub/docs/96abvalid.ps.gz.

Menzies, T. [1997*a*], 'OO patterns: Lessons from expert systems', *Software Practice & Experience* **27**(12), 1457–1478. Available from http://www.cse.unsw.edu.au/~timm/pub/docs/97probspatt.ps.gz.

Menzies, T. [1997*b*], Qualitative causal diagrams for requirements engineering, *in* 'The Second Australian Workshop on Requirements Engineering (AWRE'97)'.

Menzies, T. [1998], 'Towards situated knowledge acquisition', *International Journal of Human-Computer Studies* **49**, 867–893. Available from http://www.cse.unsw.EDU.AU/~timm/pub/docs/98ijhcs.

Menzies, T. & Compton, P. [1994], *A Precise Semantics for Vague Diagrams*, World Scientific, pp. 149–156. Available from http://www.cse.unsw.edu.au/~timm/pub/docs/ai94.ps.gz.

Menzies, T. & Compton, P. [1997], 'Applications of abduction: Hypothesis testing of neuroendocrinological qualitative compartmental models', *Artificial Intelligence in Medicine* **10**, 145–175. Available from http://www.cse.unsw.edu.au/~timm/pub/docs/96aim.ps.gz.

Menzies, T. & Cukic, B. [2000], 'Adequacy of limited testing for knowledge based systems', *International Journal on Artificial Intelligence Tools (IJAIT)* . (to appear).

Menzies, T., Cukic, B., Singh, H. & Powell, J. [2000], 'Testing indeterminate systems'. ISSRE 2000 (to appear).

Menzies, T., Easterbrook, S., Nuseibeh, B. & Waugh, S. [1999], An empirical investigation of multiple viewpoint reasoning in requirements engineering, *in* 'RE '99'. Available from http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-009.pdf.

Menzies, T. & Michael, C. [1999], Fewer slices of pie: Optimising mutation testing via abduction, *in* 'SEKE '99, June 17-19, Kaiserslautern, Germany. Available from http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-007.pdf'.

Menzies, T. & Waugh, S. [1998], On the practicality of viewpoint-based requirements engineering, *in* 'Proceedings, Pacific Rim Conference on Artificial Intelligence, Singapore', Springer-Verlag.

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

Newell, A. [1982], 'The Knowledge Level', *Artificial Intelligence* **18**, 87–127.

Newell, A. [1993], 'Reflections on the Knowledge Level', *Artificial Intelligence* **59**, 31–38.

Ng, H. & Mooney, R. [1990], The Role of Coherence in Constructing and Evaluating Abductive Explanations, *in* 'Working Notes of the 1990 Spring Symposium on Automated Abduction', Vol. TR 90-32, pp. 13–17.

Nuseibeh, B. [1997], To be *and* not to be: On managing inconsistency in software development, *in* 'Proceedings of 8th International Workshop on Software Specification and Design (IWSSD-8)', IEEE CS Press., pp. 164–169.

Nuseibeh, B. & Russo, A. [1999], Using abduction to evolve inconsistent requirements specifications, *in* 'Proceedings of ICSE-99 Workshop on Software Change and Evolution (SCE'99), LA, California, USA'. Available from `ftp://dse.doc.ic.ac.uk/dse-papers/viewpoints/nuseibeh.sce99.pdf`.

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

Pearl, J. & Korf, R. [1987], 'Search techniques', *Ann. Rev. Comput. Sci. 1987* **2**, 451–67.

Perry, D. [1999], Some holes in the emperor's reused clothes, *in* 'Proceedings of WISR9: The 9th annual workshop on Institutionalizing Software Reuse'. Available from `http://www/umcs.maine.edu/~ftp/wisr/wisr9/final-papers/Perry.html`.

Poole, D. [1985], On the Comparison of Theories: Preferring the Most Specific Explanation, *in* 'IJCAI '85', pp. 144–147.

Poole, D. [1990], 'A Methodology for Using a Default and Abductive Reasoning System', *International Journal of Intelligent Systems* **5**, 521–548.

Reggia, J., Nau, D. & Wang, P. [1983], 'Diagnostic Expert Systems Based on a Set Covering Model', *Int. J. of Man-Machine Studies* **19**(5), 437–460.

Rich, C. & Feldman, Y. [1992], 'Seven layers of knowledge represeentation and reasoning in support of software development', *IEEE Transactions on Software Engineering* **18**(6), 451–469.

Sahlin, D. [1991], An Automatic Partial Evaluator for Full Prolog, PhD thesis, The Royal Institute of Technology (KTH), Stockholm, Sweden. Available from `file://sics.se/pub/isl/papers/dan-sahlin-thesis.ps.gz`.

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

Schreiber, A. T., Wielinga, B., Akkermans, J. M., Velde, W. V. D. & de Hoog, R. [1994], 'Commonkads. a comprehensive methodology for kbs development', *IEEE Expert* **9**(6), 28–37.

Simon, H. [1969], *The Science of the Artificial*, MIT Press.

Smythe, G. [1989], 'Brain-hypothalmus, Pituitary and the Endocrine Pancreas', *The Endocrine Pancreas* .

Waugh, S., Menzies, T. & Goss, S. [1997], Evaluating a qualitative reasoner, *in* A. Sattar, ed., 'Advanced Topics in Artificial Intelligence: 10th Australian Joint Conference on AI', Springer-Verlag.

Weyuker, E. [1993], 'More experience with data flow testing', *IEEE Transactions on Software Engineering* **19**(9), 912–919.

Zowghi, D., Ghose, A. & Peppas, P. [1996], A framework for reasoning about requirements evolution, *in* 'Proceedings of the 4th Pacific Rim International Conference on Artificial Intelligence (PRICAI96), Cairns, Australia, August'.