Towards Predictive Models of Technology Impact on Software Design Productivity

Michael R. Lowry
NASA Ames Research Center
Moffett Field, CA 94035
+011 (650) 604-3369
Michael.R.Lowry@nasa.gov

ABSTRACT

In order to advance software engineering research, agencies should fund pilot studies for calibrating software design productivity impacts of potential technology advances. We need a *predictive* model of technology impacts in order to advocate technology programs and to select individual projects that provide most benefit to society. Current software cost estimation models can provide a starting point, but in the long run are inadequate because they are based on current methods and technologies for software development. Ultimately, the predictive models need to be rooted in fundamental factors affecting productivity, ranging from cognitive facility of different programming language paradigms, mathematical underpinnings for reuse and compositional approaches, and organizational psychology for large development projects. Such a productivity model would enable development of metrics for individual facets of software design productivity, and an understanding of how even narrow technology advances contribute to overall software design productivity.

Categories and Subject Descriptors

D.2.9 Software Engineering Cost Estimation, D.2.2 Software Design Tools and techniques, D.2.4 Software Verification

General Terms Economics, Measurement, Reliability

1. Overview

The benefit to society of software engineering research is principally to enable the cost-effective development and maintenance of reliable software-based systems that will enrich our lives. However, in contrast to physics-based engineering disciplines, software engineering is a young field with an immature research agenda and many difficulties in connecting basic research to improvements in practice. As an example of predictable technology impacts in physics-based engineering disciplines, consider NASA's current reformulation of the human space program. The explicit goal is developing a suite of technologies to eventually enable human Mars surface missions, with a twenty to thirty year time horizon before the first mission. This Mars mission goal is extremely costly with current technology.

The space transportation costs by themselves are daunting - even a modest mission profile placing 50 metric tons on the Martian surface (sufficient for human exploration until the planets realign and subsequent Earth return) requires approximately 500 metric tons being placed into Low Earth Orbit (LEO). Almost all this mass is rocket fuel for successive mission phases starting with the departure from LEO. This is more mass than the International Space Station that has required more than a decade of space shuttle launches. The laws of physics, especially the rocket equation, provide a means of calculating the potential impact of new hardware technologies on the mass needed in LEO, and hence mission cost. There is high confidence in calculating potential mass reduction by spending research dollars on technology goals such as ion propulsion, alternative means for supplying power, inflatable habitats for space and extra-terrestrial exploration, and in-situ production of rocket fuel on Mars. Note that all these are disruptive in terms of their effect on mission profiles compared to current technologies. New space transportation technologies that have yet to be imagined can also be assessed using the same basic physics for their impact on reducing mass needed in LEO for a Martian mission, and other attributes that directly translate to cost and reliability.

A fundamental question is whether such a predictive model for software engineering productivity based on first principles is possible. The impact of software engineering technology advances is usually considered too difficult to predict, especially for disruptive technologies that could fundamentally change the way software is engineered. In part this is because software is principally a design problem; costs are almost exclusively for human engineering labor incurred during development and maintenance. Operational benefits such as system capability, reliability, and safety are considered too difficult to predict and quantify; hence cost-benefit decisions face large uncertainties. High levels of uncertainty have a profound effect on the time value of money, and hence investment decisions. Research investments in physics-based hardware technologies have the advantage of reliable predictions of potential benefits; investment decisions can thus focus on risk factors such as whether a technology goal can be achieved – and management issues such as investing in one or multiple approaches, and trade-offs in investing in technologies with higher potential benefits but greater risks in achieving a technology goal. Investing in software technology advances is thus problematic within a wide portfolio of research dollars for competing technology disciplines, since calculating even the benefits is highly uncertain.

This paper argues that even predictive models based on rough empirical correlations can clarify research investment decisions. Furthermore, as these rough predictive models undergo increasingly precise empirical validation, they could lay the groundwork for models based on first principles – much as the Ptolemaic theory led to Newton's theory under the increasingly precise empirical observations of telescopes. Predictive models of software engineering technology impacts would provide a firm foundation for increased software engineering research investment within a portfolio of competing technology disciplines. They will also be informative for selecting and managing a portfolio of software engineering research projects, much the same way NASA is using (multiple) Mars Design Reference Mission concepts to evaluate synergistic hardware technology benefits as well as tall-pole requirements. Finally, these models will be informative to the software engineering research community, and might stimulate new directions for research.

There are two principle ingredients for a predictive model of technology impact. The first are uncalibrated mathematical laws that predict outputs – especially productivity (cost per unit of software) and reliability (latent defects that manifest during operational use), as a function of factors that could be effectively impacted by technology advances. These mathematical laws could at first be based on qualitative factors without a first principles underpinning, such as shown in section 3, and then evolve. The second are experimental studies that provide empirical evidence for the validity of these laws, and calibrations of parameters. Existing software project cost models can provide a starting point, especially with scaling laws that can be qualitatively related to selected technology advances, illustrated in section 3. However, the purpose of software cost models is inherently different: to predict the cost of specific software projects based on a wide variety of factors most of which are only peripherally related to technology variations. In addition, while current cost models are well calibrated, their calibration is necessarily against a backwards-looking corpus of projects that typically span decades.

A principle impact for society of software technology advances will be to reset the trade-off curve between productivity and reliability. Within a given level of software technology, in order to obtain increased reliability of a software system, more human labor for assurance activities is required thus decreasing productivity. Technology advancement becomes a set of contours relating productivity to reliability, for example by automating portions of assurance activities. Future technology advances might also change the shape of these contours. One characteristic is that even for casual software projects, post-deployment defect management is costly. Technology advances for effective early-lifecycle defect reduction can lower overall costs. In other words, maximal productivity for casual yet functional software is achieved with a large but bounded defect density – densities beyond this bound increasing post-deployment cost and hence lowering productivity, while defect densities less than this bound are correlated with assurance activities that cost more than optimal for the casual nature of the software. Technology advances for effective and low-cost early-lifecycle defect reduction can lower this optimal bound and change the shape of the reliability versus cost contour.

Predictive models of software technology impacts will need to be based on fundamental factors in order to extrapolate outcomes such as changing contours of productivity versus reliability based on inputs that are different mixtures of technology advances. For example, currently there is no reliable way to predict how different advances in code-based static analysis would interact with different technology advances in model-based software engineering – and how these in combination or separately relate to targeting massive multi-core hardware architectures. In addition, technology advances that fundamentally change the practice of software engineering – such as widespread end-user programming, have impacts that can only be modeled using fundamental factors. Section 4 describes a starting set of fundamental factors. The conclusion of this position paper advocates steps that government agencies can take to calibrate predictive models.

2. CRISIS IN SOFTWARE DESIGN PRODUCTIVITY

Transistor-based computers reliable enough for routine use became available for commerce fifty years ago (1960s). By the 1970s general-purpose computers had begun to revolutionize the work life of the small portion of the population that had routine access to the Arpanet and time-shared computers. Within a generation, software-based systems transformed work and home life throughout the developed world. Work is centered on creating and communicating digitized documents or controlling computerized equipment; communication is through 24/7 mobile phones and high-bandwidth Internet, entertainment through increasingly interactive on-demand digitized television.

Whither the software crisis? It was in the context of the early decades of the software revolution that the critical problem with software design productivity became evident – in essence, the difficulty of writing correct, understandable, and verifiable software. The term, coined at the 1968 NATO Software Engineering conference [5], was elaborated by Dijkstra in his ACM Turing Award lecture [3]: "now that we have gigantic computers, programming has become an equally gigantic problem". The crisis is manifested by software projects that are over-schedule and over-budget – if delivered at all - with low-quality and unmaintainable software. There has been incremental progress but no silver bullets for the 'software crisis'. The crisis is not in our ability to produce software, but in our ability to produce reliable and verifiable software to standards that can be used in critical applications without excessive cost.

In the commercial realm, the crisis is usually sidestepped. The mass distribution of software-controlled devices amortizes large software development budgets, which are further ameliorated through outsourcing to developing countries. The software quality is seldom required to be safety-critical or even mission-critical. Personal computer software is driven by time-to-market, with quality and correctness a secondary factor. In other cases, such as the automobile industry, government oversight has not kept pace with the profound transformation from mechanical/electrical analog control to digital control. Hence even safety-critical software is often developed with standards that are not demonstrably commensurate with the risks involved.

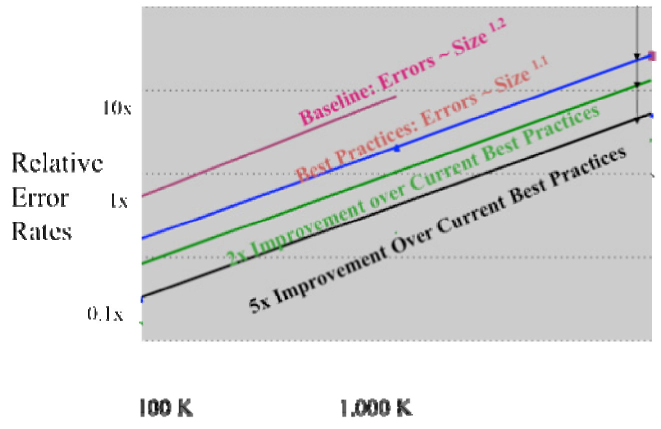
The challenge to the research community is technology for the cost-effective development of reliable software for new systems that will continue to enrich our lives: cars that safely drive themselves, space systems for human exploration of Mars, renovations of our National Air Space that provide expanding capacity without compromising safety, and sophisticated medical devices that extend our lives and health. The methods and technologies for the development of these future systems is the concern of the inter-agency software design and productivity co-coordinating group [3] that co-sponsors this workshop.

3. QUALITATIVE ESTIMATES OF TECHNOLOGY IMPACTS FROM EXISTING COST MODELS

For safety-critical software, defect management is the major determinant of software cost. Existing software cost estimation models can provide rough order of magnitude scaling laws of software cost versus software size, which when combined with defect prediction can yield qualitative extrapolations for technology impacts. As discussed later, this method is limited because cost models and defect models are highly correlated through calibration on backward-looking projects. Here we illustrate the methodology by considering extrapolations made for NASA's unmanned space exploration program a decade ago [4].

Historical data shows aerospace software size increasing substantially over the past decades, for both manned and unmanned missions. Software cost models such as COCOMO [1] have calibrated scaling models where software cost increases proportionally to $(\text{size})^{1<N<2}$. This scaling law assumes that the number of software modules increases linearly with overall software size. Software assurance activities for large-scale software systems focus on unintended interactions between modules, such as verification at software system integration. Because they are not local, defects due to these unintended interactions are costly to resolve. The number of interactions is bounded from above by the square of the number of modules. Thus logically N varies between 1 and 2. Calibration on cost and schedule data from past software projects gives a value of N=1.2. In these models cost, schedule, and errors are all correlated.

Given the calibration of N; the correlation between cost, schedule, and defects, and the historical growth of software on aerospace systems we can extrapolate to future missions. A calibration of defects from past unmanned Mars missions was used to extrapolate the defects for future missions, using N = 1.2 . Mars missions are notoriously difficult; internationally two out of three missions have failed. At the end of the last decade, failures of Mars Climate Orbiter and Mars Polar Lander – both most likely due to software-related errors – led to the calibration in the log-log graph below; which indicated that the long-term prospects for reliable Mars mission software were poor.



The relative error rate vertical axis on this log-log graph indicates errors that are both mission-critical and that have high probabilities of being executed, such as occurred on Mars Polar Lander. With the calibration to the Mars Polar Lander software size, the extrapolation of the red line is that the number of mission-catastrophic errors for the expected software size for a sample return mission to Mars would be over ten – hence little chance of success. (In a Mars sample return mission, a robotic vehicle collects samples then returns to Earth.) This qualitative model demonstrated a need for improvement in software design methodology, possibly incorporating technology advances.

Best practices such as CMMI tend to control unintended interactions, reducing the size of the exponent N. The blue line indicates that even reducing N to 1.1 gives a reasonable chance of mission success for software in the 200K range, but is insufficient by itself when scaling to a million source lines of code.

Ten years ago model-based software engineering methods were being first used in practice. The approach is to develop an abstract model of both the software and its domain of operation. These models make interactions explicit and precise, thus enabling interactions to be understood at systems integration level. The level of abstraction reduces details of the interactions and focuses attention on critical attributes. Over the last decade this approach has been increasingly adopted in the aerospace community. Although the full set of capabilities envisioned in research laboratories have not yet been realized in industrial practice, cost calibrations on actual projects have already shown a 1.5x improvement, which is a significant portion of the 3x improvement on the green line.

Additional scaling laws from software cost models can sometimes be directly applied to advances in software processes or software development technology. Most cost models have a direct scaling of software size and hence cost based on language level. With other factors held constant, a human programmer takes the same time to develop a line of assembly code as a line of a domain-specific language. Due to the economy of expression in the domain-specific language, the size of manually developed software decreases proportionally, as well as the number and hence interaction between modules – even as the size of generated source code remains constant. This is part of the 5x improvement extrapolated in the black line, for which there is a reasonable chance of mission success for the software size anticipated in a Mars sample return. It should be noted that extrapolating the effect of higher-level languages and autocoding has many uncertainties. For example, the initial outside cost projection for NASA's Orion capsule had to be redone partly due to this factor.

4. TOWARDS CAUSAL MODELS

As demonstrated above, the scaling laws inherent in current software cost estimation models can provide qualitative guidance on the impact of software technology advances.

However, as predictors of the productivity impact of technology advances the models are inherently limited. First, of necessity, software cost models are calibrated on backwards-looking projects. Portions of the parameters are antiquated. One example from a commercial cost estimation tool is a parameter that rates (high – medium – low) the time it takes for a software development environment to respond to a keyboard input. This parameter was highly relevant when software development environments were run on remote time-shared computers. Today it is considered irrelevant by most cost analysts, but it is kept in the model for backwards compatibility and because it was measured in the software projects on which the model was calibrated. These antiquated parameters are interesting to consider from the viewpoint of more general models, such as the productivity of an individual programmer working in an IDE, and are one indication of the need for more fundamental factors.

Second, in these models software cost and schedule are highly correlated with software defects. If you try to compress the schedule, then defects and costs go up. To reduce predicted defects, a cost analyst raises the level of assurance-related parameters, which then entails that costs go up – accounting for the current high premium for costs of safety-critical aerospace software versus commercial software.

However, successful technology advances in software engineering will cause a *divergence* in this trade-off curve between software costs and software defects – costs will go down while defects will be held constant. Other than the scaling laws illustrated above, the cost models don't indicate underlying mechanisms for the impact of technology advances. While software costs models typically do have a parameter that represents use of more advanced technologies, such as formal methods, the calibration of the impact of this parameter is subjective or anecdotal.

A software cost and defect model that is able to extrapolate the impact of technology advances on software design productivity will need to be based on fundamental factors. Some of these are inherent in organizational psychology, others in the causal factors for individual design productivity, and others yet in the mathematical nature of software defects and models of their propagation, detection, elimination, and mitigation. Below some of these fundamental factors are described in the context of software development where defect control is critical:

Defect introduction and propagation: the cost of fixing defects as they propagate from one phase of the software lifecycle to the next phase is empirically known to grow exponentially. The mechanism is qualitatively understood, but needs to be calibrated. Boehm's CoQualmo model [2] provides a simple defect introduction/propagation/elimination framework that can serve as a starting point if combined with software rework models. As defects propagate through the lifecycle they interact with subsequent design work, thus leading to an expanding area of infection. When the defect is detected, this infected area needs to be reworked. Existing cost models typically have rework submodels for reuse and maintenance, which can serve as a starting point together with CoQualmo for calibrating defect removal cost models.

Feedback loops: outside of a single end-user programmer developing code for her own use, miscommunication is a major source of defect introduction. Requirements analysts often misunderstand end-user needs. Technology advances such as rapid prototyping environments or executable specifications potentially provide means of closing the feedback loop between analysis and end-users that is likely superior to natural language dialogue. Similarly, miscommunication across a large software development organization or between successive stages of software development is a primary source of defect introduction. A control-theoretic feedback-loop framework overlaid on CoQualmo could potentially provide a first principles model – spanning from new approaches for requirements development to new approaches for code review. A control-theoretic model could be used to simulate the value of a new technology before it is actually developed, and calibrated by tracking defects as error signals on existing software projects.

Reuse: technology and methodology advances – ranging from software product lines, to design patterns, to aspects, can profoundly impact the degree of reuse. Reuse can be done at many different levels – from requirements through code to test. Fundamental factors include cognitive models for selecting reusable artifacts, cost models for generalizing an artifact so that it is reusable, and defect models related to inappropriate reuse.

Defect detection: software verification technology can have a profound impact on cost through both enhanced, and earlier defect detection, as well as replacing human labor in software assurance activities. However, due to the expertise required to use advanced verification technology, it is currently difficult to calibrate the effectiveness of different software verification technologies. Controlled studies at universities or research labs with a suitable pool of expert users could provide data points enabling extrapolation. Advanced verification technology is an example where the impact of a technology will change over time because of the transient effects of the learning curve; and how this learning curve can be factored out with well-chosen control studies.

Individual designer/programmer productivity: the factors that lead to individual productivity such as development environment technology, programming language features, and design model formalisms, are only partially understood.

Many of the factors described here have been considered elsewhere, but have not yet been integrated into a model that can predict the software design productivity of technology advances. Further work is needed in developing the fundamental mathematical laws and integrating them, but most importantly experimental studies are needed to empirically validate the laws and calibrate an integrated model.

5. CONCLUSION

Empirical validation and calibration of this proposed model is critically required for even rough order of magnitude extrapolations predicting the impact of software technology advances. There are several possible venues for empirical studies that can be fostered by both government and industry. University software engineering practicums with multiple student design teams are now widespread, and under the careful supervision of diligent professors can yield useful qualitative or even controlled data. The government can also use its own small software development projects as experimental vehicles for new technology through supplemental funding or shadow projects. Large government-sponsored software development projects typically use geographically dispersed teams tied together electronically with all facets of the software lifecycle recorded in databases. There is a wealth of data to be mined; especially when a contractor proposes advanced technology as part of the project.

This type of predictive model for technology impacts based on fundamentals and empirically validated, will benefit both society and the software engineering research community. First, it will clarify the return on investment of software technology research, thus providing a firm foundation for increased software engineering research investment within a portfolio of competing technology disciplines. Second, it can inform the selection and management of portfolios of software technology research. Third, this type of predictive model can stimulate new ideas in software engineering research, perhaps leading to revolutionary advances.

6. REFERENCES

[1] Boehm, B., et al. 2000. *Software Cost Estimation with COCOMO II.* Englewood Cliffs, NJ: Prentice-Hall.

[2] Chulani, S., Boehm, B. 1999. *Modeling Software Defect Introduction Removal: COQUALMO (Constructive QUALity Model).* Technical Report USC-CSE-99-510. University of Southern California.

[3] Dijkstra, E. W. 1972. The Humble Programmer. *Communications of the ACM* 15 (10): 859-866. 1972 ACM Turing Award lecture.

[4] Green, C., Lowry, M., Norvig, P. 1999. Towards Reliable Mars Mission Software. Internal NASA Report.

[5] Nauer, P., Randell, B. (eds) 1969. *Software Engineering: Report of a conference sponsored by the NATO Science Committee; Garmish, Germany October 1968.* Brussels, Scientific Affairs Division, NATO 231 pp.

[6] Porter, A,. Sztipanovits, J. (eds) 2001. New Visions for Software Design and Productivity: Research and Applications. Vanderbilt University Technical Report on the Workshop of the Interagency Working Group for Information technology research and Development (ITRD) Software Design and Productivity (SDP) Coordinating Group.