Software Engineering as Technology Transfer

Daniel Cooke and Jason Denton
Department of Computer Science and the Center for Advanced Intelligent Systems
Texas Tech University
{daniel.cooke}, {jason.denton} @coe.ttu.edu

Abstract

We propose, as a challenge to the software engineering research community, a new view of software engineering as technology transition. In this view, the role of software engineering is to take the initial results of theorists and harden these results to the point where they are ready to be used in fielded, commercial systems. Current techniques for software verification, while adequate for serial, non-critical systems, are not sufficient to deal with verification and validation of parallel, concurrent, and embedded systems. We examine how the software development process might be modified by this view of software engineering as a process to migrate new technology to systems ready for use in high reliability environments.

Keywords: Technology Transition, Software Systems Hierarchy

1. Introduction

In recent years many people in the software engineering research community have been devoted to formulating models for the evaluation and improvement of the development process of large scale software projects [7, 15]. This work has largely focused on applications to meet and support various business and information processing needs; systems which are largely serial and relatively non-critical in nature. Such systems and the techniques for building them are fairly well understood by software engineers. There remains a need, however, for techniques to transition new software technology from laboratory products to robust, delivered systems. Here, we propose a more comprehensive approach to software engineering that addresses this need for technology transition. We model this approach on the technology readiness levels used by NASA [14] and attempt to incorporate the following considerations and influences, relevant to software development, in the challenge we propose:

- The impact of *Technology Transition*;
- The impact of a hierarchy of *Validation and Verification* requirements for differing classes of problems;
- The impact of *teaming and compartmentalization* on a development project; and
- The impact of *Software Architecture*.

2. The Technology Transition Context

NASA defines nine technology readiness levels, spanning the range from basic research to systems which are ready for launch and flight [14], as shown in figure 1. Lab products or prototyped systems and initial theoretical results reside in the first three technology readiness levels. At levels four through six products are developed and matured, becoming capable of addressing a wider class of problems of interest to problem domain experts. As technology reaches the point where it can be built into production quality flight systems and sub-systems it enters TRL seven, becoming ready for flight and launch at levels eight and nine.

Roughly speaking, one can project three basic regions of technology readiness. As an example, consider some recent advances in logic programming. Recently, theorists focused on logic programming advanced an important result, called action-based reasoning [10]. The initial result by Gelfond and Lifschitz was able to show progress on problems of theoretical interest to the logic programming community. This initial result is a lab product, residing in the first region of technology readiness. A lab product does not approach production quality. Efforts to transition the results have resulted in a mid level product (in the region from 4-6) that can be applied to more elaborate problems; problems of interest to people working in a particular problem domain. In the case of action-based reasoning, the result has undergone significant hardening to the point that it is now being used to discover work-around's in the event of some subsystem failures on the Space Shuttle [2]. These work-around's

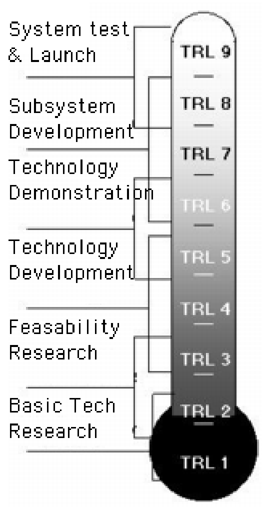

Figure 1. NASA Technology Readiness Levels

are normally discovered by Mission Controllers in Houston. These results have gained the interest of problem domain practitioners. If the resulting technology demonstrations are compelling enough, efforts to further harden the result to production quality will be undertaken. Once hardened systems are developed to support mission controllers, they will be analyzed to determine if subsequent versions can be deployed on long distance missions, where substantial communication delays make communication with mission controllers impossible.

In the case of the action language result, one can see deliverables at three points. After basic research there is a lab product that can demonstrate solutions to problems that are of import to the theoretical community. Transition to the mid-TRL range hardens the product to the point that it is possible to perform technology demonstrations within a problem domain community. Transitioning to the highest technology readiness levels results in a production quality system; one that at NASA is flight ready.

The ideas put forth here extend the initial ideas of Joint Application Development to include, in the development teams of software engineers and problem domain experts, researchers who have developed fundamental results applicable to a problem domain.

These ideas also attempt to encompass the fact that there are indeed different classes of software. These perceived classes are based upon attendant validation and verification requirements and should result in a hierarchy of software process models. In fact a projection of these classes based upon the regions of technology readiness is conceivable. For example, the proofs-of-concept or prototypes arising as lab products in regions 1-3 of technology readiness have very low verification and validation requirements. One can envision ground-based systems that are serial in nature in the 4-6 regions and parallel/concurrent flight ready systems in the 7-9 regions. It should be noted that the transition path for a flight ready system would need thorough ground testing before going to flight readiness. In the case of the action language work, the 4-6 testing is taking place in mission control - and the tested system is augmenting and supporting their efforts. Once confidence is assured, the systems will be tested for flight readiness, beginning with simulations, proceeding to experiments on unmanned platforms, and ultimately to deployment in manned spacecraft.

3. Joint Application Development with an Accent on a Scientific Approach

People performing research in most of the physical sciences will normally classify themselves as theorists or experimentalists. Most communities are organized in this manner because the exchanges between theorists and experimentalists are the basis for scientific progress. Experimentalists present observations, which theorists attempt to condense into theories. The theories are supposed to provide an understanding of the root causes for the empirical data, and if correct, allow for prediction of future phenomena. These predictions, in turn, provide the basis for the design of future experiments. Experiments result in confirmation or refutation of the theories. Improvements are made to existing theories when observations are not explained by a theory. Revolutionary advances occur when a new theory results in a paradigm shift. Having an experimental counterpart to a theoretical community is crucial for advancing a science.

Computer Science lacks a pervasive, organized experimental community; theories are rarely tested empirically. The notion advanced here is that hard application domains are a fine substitute for experiment. When competing theoretical approaches to a class of problems are developed into more robust systems, they can be tested against a relevant hard application and provide technology demonstrations. The best application results can provide feedback to the theoretical communities, the exchange resulting in advances otherwise difficult to accomplish.

To take an idea from a proof-of-concept lab product and actually test it against a hard application requires a software engineering effort that will result in a more robust system. We believe the notion of Joint Application Development should be expanded to include more than problem domain experts. The theorists who developed the approach or idea that is being taken to product should be included, resulting in *group application* development. This is not a new idea. Years ago, Richard Feynman, when working at Los Alamos, was dispatched to Oakridge, where engineers were building the plants to produce the materials for the atom bomb. The engineers needed to be briefed on the theoretical aspects and context within which they were working. After the briefing, the engineers were able to correct serious problems in their initial designs and ultimately construct the plants that served a major role in winning World War II [9]. This kind of group application development is essentially experimental design. Thus, software engineering as technology transition holds the potential of transforming computer science - software engineering might eventually provide the pervasive, organized experimental counterpart to theoretical efforts in computer science.

4. Process Models: One Size Does Not Fit All

The Group Application Development approach to software engineering requires a different process and process-supporting tools if it is to transition theoretical ideas to a higher level of technology readiness. In order to be effective, process models must be tailored to their environments and the goals of the developing team. Motorola credits process based improvement with greatly reducing the time to market for new products [6], when used for that purpose. For transitioning new technologies to higher readiness levels, process models should focus on verification and validation.

Software can be broken into a hierarchy of minimally three different classes systems. As systems move up this hierarchy the verification and validation requirements expand; the verification and validation requirements for a class of system are always a superset of those required by systems at lower levels of the hierarchy.

4.1. Level 1: Serial, non-Critical Systems

Development of serial non-critical systems has been extensively studied. These system include most shrink wrapped software and data management software for businesses. Most of the literature on process models has focused on the specification and coding of such systems. Verification of the code is done with respect to the specification. Clean room [13] and statistical approaches [3] to testing software seem to suffice for software verification of these types of systems.

4.2. Level 2: Parallel and Concurrent Systems

The problem of software quality assurance is made more complicated by concurrent or parallel program structures. Problems like race conditions and deadlock are unique to the concurrent programs. The likelihood of such defects being observed in any given run of the program may be very low.

Traditionally, software verification is accomplished with respect to a specification of the software. If sufficiently precise and complete, the specification serves as a mathematical model of the intended consequences of the software, given its field of inputs. Typically, formal verification of a program is not feasible, so statistical approaches to software testing are utilized. Mathematical models used to determine if a program is correct, whether by testing or by proof, are also necessary for the verification of concurrent programs. However, current approaches taken alone are not sufficient for verification of concurrent programs.

The problems of concurrent design were pointed out in Leveson's study [12] of the famous Therac 25 accidents. Although the Therac software was tested, a race condition was not detected. The race condition eventually occurred when the machine was put into use. The Therac 25 administered the doses to patients undergoing radiation treatments. Several of these patients died after receiving overdoses of the radiation.

Consider the following example where two different processes or threads alter the information stored in the shared variable $dose$.

t_1	t_3
t_2 : write $dose$	t_4: write $dose$

There are six possible orderings for the execution of the instructions in these two paths.

$t_1t_2t_3t_4$
$t_1t_3t_2t_4$
$t_1t_3t_4t_2$
$t_3t_4t_1t_2$
$t_3t_1t_4t_2$
$t_3t_1t_2t_4$

Suppose instruction t_4 writes the correct dose for a treatment. In the configuration given there is a fifty-fifty chance that an erroneous condition, where t_2 writes the final value of dosage, will occur. Testing alone may not cause the error to occur.

Sequential systems are typically deterministic, every execution of the program on the same data performs the same computation. Nondeterminism occurs in concurrent systems because different executions of the system may result in different computations if instructions in different execution paths are executed in a different order.

When concurrent pathways of execution exist, there are a number of possible orderings for execution. For the sake of simplicity, assume all execution paths are of the same length. Let there be n execution paths each of length L. Then, the number of possible orderings for execution of program O is given by

$$O = \frac{(nL)!}{(L!)^n} \tag{1}$$

The number of orderings increases dramatically with increases in either n or L. For example, when $n = 3$ and $L = 2$, $O = 90$.

Due to the nondeterminism in concurrent systems, problems like race conditions, deadlock, and starvation may not be uncovered by testing alone. The orderings in which the problems occur may not arise during the testing phase. To fully scrutinize a concurrent program, one must test and model the system. Often a separate engineering model which focuses on some phenomena or aspect of the system is required to analyze a program.

In the case of a concurrent program, the phenomenon to be understood is the control and data flow view of the program. Through this view, one can gain an understanding of the concurrent behavior of the program in question. Petri net modeling of concurrent programs provides a suitable approach to the discovery of problems that are unique to concurrent programs [1, 4, 8].

4.3. Level 3: Embedded-Critical Systems

The NASA Ames Research Center has led in the development of a neural network-based Intelligent Flight Control system [11]. To simulate the loss of an entire wing, the test flight airplane can position an airfoil in front of the main wing surface. This carefully positioned airfoil creates turbulence that renders the main wing ineffective, as if it were removed from the fuselage. When the IFC software is enabled, the pilot can regain control of the aircraft. The IFC system is a good example of model-based automated reasoning based upon a model of flight. The system can integrate into a fly-by-wire aircraft and learn its flight characteristics through observation of pilot inputs and aircraft response. In doing so, the system exemplifies intelligent data understanding through its ability to establish and learn the causal links between inputs and aircraft response. In the future, such systems could be used to make civilian and commercial aircraft safer and more resistant to catastrophic failures. The IFC software is a good example of a class of systems that require all the previous verification and validation steps, but also involves extensive simulation and ultimately flight testing. These activities are a part of the verification and validation process, and must be taken into consideration by the software development process. This process is further complicated by the fact that the IFC software adapts itself to its changing environment.

5. Teaming and Compartmentalization

Complex problems are best solved by simple and elegant solutions. Consequently, successful software solutions often require invention and innovation. Much of the current thinking and literature in Software Engineering recommends the compartmentalization of the process of developing and in the substance of software solutions, together with a division of labor that results in team efforts for software development. But, is this thinking correct in general, or only in particular?

The compartmentalization of the substance of a software solution arises from the subdividing of a problem into component or module-sized pieces considered to be more approachable for a programmer. The modules to be developed are identified early in the project and bias the solution towards an approach that is reached at a time when there has been little opportunity to understand the problem in a manner that permits multiple approaches to be considered. The compartmentalization of the substance of a software solution is a response to a rather short-term view of the costs associated with software development and an overriding managerial concern for cost containment. Does this view of software development distill the process to the point that more elegant, simpler, and more maintainable solutions are less likely to be found? Are there different processes that would enhance a project's opportunity to invent better solutions?

Ken Thompson points out that the efforts with which he has been involved are characterized by small groups of individuals who are technically savvy and knowledgeable about the software solution in total [5]. Thus, all members of the group are free to offer their own approaches to the solution and approaches can be challenged and discussed.

The compartmentalization of the process of software development often serves to discourage developers from writing code early in the process. The main exception to this tendency occurs when developers prototype the solution in the requirement or specification phase in order to validate the solution with the client. Thompson points to a different process where different group members program proofs of concept in order to advocate their respective technical approaches to the solution [5]. In other words, when two members of the group differ in their technical approaches to the problem, they write programs that demonstrate their approaches so that their views can be compared in a practical manner. These programs are written very early in the process so that technical assumptions about the design can be resolved and design can go forward with less backtracking in future phases of the project.

The teaming currently advocated for software projects arises from the concern that the problems now being solved are too complex for a single individual to comprehend, and the total solution can exist only as the sum of the knowledge held by the members of the team. Thompson is quick to point out that at Bell Labs there are no teams and no leaders [5]. There are only groups of individuals interacting with one another. Apart from the issues already noted about compartmentalization in teams, there is a philosophical concern about the subjugation of the individual to the collective, and the belief that innovation typically arises from individuals not groups. Obviously Bell Labs is a unique place and one may seriously argue that what works there may not work elsewhere. However, curiosity leads one to wonder whether the process makes the individuals. If more projects were conducted in a process that freed individuals to be more creative, perhaps similar results would be achieved elsewhere.

6. Software Architecture

Another goal related to the technology transition process and the associated validation and verification is the development of a more general approach to software architecture. It appears that many, if not most, software applications can be developed by building a system from the inside out. By this we mean that an application should be reduced a small set of primitives from which one can compose a *transaction core*. Designing a system in this manner requires a signicant amount of thought, but facilitates a simpler and more elegant solution to a software application. The anti-teaming and compartmentalization organization is a critical supporting influence on achieving the desired results of a primitive-based application system.

After designing the primitive *transaction core*, the design and implementation group can then go about building the interfaces for the system. The interfaces may be with other software systems, controlled instruments (in the case of realtime, embedded systems), and/or people who are users of the system. Let's consider a simple example of this approach.

Suppose a word processing system is being developed. Envision a data structure like the cellular list below:

U	s	e	r		i	n	p	u	t

Figure 2. Word Processor Data Structure

This structure, combined with information and the current position of the cursor and mouse pointer might be adequate for the state of the system. Primitives for the system's transaction core would include *select_cells*, *stack_operations*, *remove_cells*, and *insert_cells*. These primitives could also have primitive modifiers to provide such things as font, font size, and other text modifiers like boldface, italics, and underscore.

After defining the primitives, a developer would combine some of them to compose more complex transactions. For example, after performing a *select_cells* operation, a cut transaction would be composed of a *push(remove_cells)*. Eventually the system would be composed to the point that developer can focus on the system's interfaces, answering questions such as:

How will the user select a cell?
How will the user select an arbitrary number of contiguous cells?
How will the user select an arbitrary number of non-contiguous cells?
:
How will the user view the result of an *insert_cells?*
Etc.

Once the transaction core is established, interface issues may be resolved with reusable objects or by building a GUI system, also from the inside out. In either case, the architecture must provide facility to combine elements of the transaction core, an ability to build interfaces, and the proper protocols for linking interfaces with the reansaction core. The general architecture is shown in figure 3.

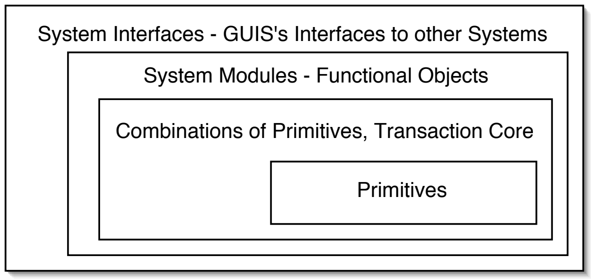
System Interfaces - GUIS's Interfaces to other Systems
System Modules - Functional Objects
Combinations of Primitives, Transaction Core
Primitives

Figure 3. General Architecture

The approach of reducing an application to its primitives requires careful thinking about the fundamental behavior of the system. It is a computer science approach to system development in that one must also consider how the primitives should interact in order to compose sophisticated operations.

This approach helps organize the process model for the full system development. The interplay among the verification and validation requirements for the application domain, the theoretical results to be transitioned to higher TRL through their embodiment in the application, and the system architecture become three important variables in process development and execution.

Such an approach can significantly impact verification, validation and system evolution. Simple components at the core of the system are easier to test for correctness, and less likely to be affected by high level design changes. This increases system stability and reliability. Having these parts of a system segregated, with clear boundaries, should minimize the ripple effects on the transaction core resulting from changes to the system interface. Furthermore, verification of the transaction core might be more intensive, perhaps involving formal verification.

7. Conclusion

Verification and validation of concurrent, embedded real time systems is a difficult task. As software comes to play a more important role in the control of vehicles and other safety critical systems, we must extend and adapt our existing process models to meet the challenges of developing these kinds of systems. Changes to the underlying architecture of such systems, and in the organization of the developing teams, can also contribute to creating an environment where initial lab products become ready for production systems. The ultimate challenge is to transition relevant fundamental research results into the software solutions in various problem domains. It is our basic premise that this oversight in software engineering research must be remedied for the sake of the computer science and software engineering disciplines.

References

[1] G. Balbo, S. Donatelli, and G. Franceschinis. Understanding parallel program behavior through petri net models. *Journal of Parallel and Distributed Computing*, 15(3):171–187, July 1992.

[2] M. Balduccini, M. Gelfond, M. Nogueira, and R. Watson. Planning with the usa-advisor. In *Proc. of 3rd International NASA Workshop on Planning and Scheduling for Space*, 2002.

[3] D. Banks, W. Dashiell, L. Gallagher, C. Hagwood, R. Kacker, and L. Rosenthal. Software testing by statistical methods. Technical Report NISTIR 6129, National Institute of Standards and Technology, Mar. 1998. http://www.itl.nist.gov/div897/ctg/stat/mar98ir.pdf.

[4] A. T. Chamillard and L. A. Clarke. Improving the accuracy of petri net-based analysis of concurrent programs. In *International Symposium on Software Testing and Analysis*, pages 24–38, 1996.

[5] D. Cooke, J. Urban, and S. Hamilton. Unix and beyond: An interview with Ken Thompson. *IEEE Computer*, 32(5):58–64, May 1999.

[6] M. Diaz and J. Sligo. How software process improvements helped Motorola. *IEEE Software*, 14(5):75–81, Sept. 1997.

[7] A. Dorling. Spice: Software process improvement and capability determination. *Information and Software Technology*, 35(6/7), June 1993.

[8] M. B. Dwyer, L. A. Clarke, and K. A. Nies. A compact petri net representation for concurrent programs. In *Proceedings of the Seventeenth International Conference on Software Engineering*, Seattle, Washington, Apr. 1995.

[9] R. P. Feynman. *The Pleasure of Finding Things Out*. Perseus, Cambridge, MA, 2000.

[10] M. Gelfond and V. Lifschitz. Representing action and change by logic programs. *Journal of Logic Programming*, 17:301–321, 1993.

[11] J. Kaneshige, J. Bull, and J. J. Totah. Generic flight control and autopilot system. Technical Report AIAA-2000-4281, American Institue of Aeronautics and Astronautics, Dec. 1999. http://ic.arc.nasa.gov/ic/publications/pdf/2000-0172.pdf.

[12] N. Leveson and C. S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[13] R. Linger. Cleanroom process model. *IEEE Software*, 11(2):50–58, Mar. 1994.

[14] National Aeronautics and Space Administration, Washington, D.C. *Integrated Technology Plan for the Civil Space Program*, 1991.

[15] M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model for software, version 1.1. Technical Report CMU/SEI-93-TR-24, Software Engineering Institute, Feb. 1993.