On Verifiability and Composability for High Confidence Software

Wu-Hon Francis Leung, Computer Science Department, IIT. leung@iit.edu

1 Several necessary conditions for high confidence

High confidence in a software system depends on how thoroughly it is verified. Presently, software verification relies on testing. But the software of a cyber-physical system is typically organized as cooperating concurrent processes that must react to triggering events and change in conditions rapidly. The execution of the processes, the messages that they exchange and the events that they receive can interleave in many different ways. The state space of such a system is exponential to these variables. Testing can only explore a minute portion of that space. Development projects already are spending a majority of their time and effort in testing, yet many errors are still found in the field. Therefore, meeting the following (verifiability) is necessary:

N1: Software development must depend more on assertion based (or formal) verification instead of instance based testing.

The problem of verification is made much more difficult because the software base constantly changes as features[1] are added to the system. Changing code is error prone and laborious. The programmer must examine large amount of code line by line. At the end, he is usually uncertain whether he has identified all the code that need to be changed and what effect his changes may have on the other features. The job will take iterations of testing and debugging. When a feature is implemented by modifying the code of other features, the programs of the features *entangle* in the same reusable program unit (e.g. a method) of the programming language. Entangled programs are difficult to maintain, verify and reuse. As more features are added to the system, the software becomes exceedingly complex decreasing programmer productivity and increasing testing effort. The following (composability) is also necessary:

N2: A feature can be developed without entangling with the code of other features.

The problem of program entanglement has severe implications in practice. For example, with existing programming languages, the programs of normal processing and exception handling features are entangled. A new exception thrown by a device driver sometimes requires manual review of most of the application code. If the project is not willing to pay the cost, the software will crashes or hanged when the exception is not caught. This author witnessed that program entanglement had degraded programmer productivity to only a couple hundred lines of code a year after a few releases. This problem cannot be solved using existing general purpose programming languages[2]. As a result, the following is also necessary:

N3: New programming language concepts that facilitate solutions to N1 and N2.

In section 2, we will elaborate on the challenges posted by N1 and N2. In section 3 we briefly describe our on going work on N3. This position paper concludes with a brief remark in section 4.

---

[1] A software project often uses the term *feature* to denote certain functionality of the system. For example, congestion control is a feature of TCP. The terms feature, service, concern and aspect are often used in the literature interchangeably.

[2] W. H. Leung, "Program entanglement, feature interaction and the Feature Language Extensions," *Computer Networks*, February, 2007.

2 Research challenges

2.1 Feature interaction is the main reason for program entanglement

The problem of program entanglement is related to the notion of *feature interaction*. Two features *interact* if their *behavior* changes when they are integrated together. Features are implemented by computer programs whose behavior is manifested in its execution flow and output for a given input. The term feature interaction was introduced by developers of telecommunications, but its occurrence is common place. For example, adding exception handling changes the behavior of existing software from crashing when unusual events occur. We show in [2] that if (C1) two features *interact*, (C2) they are executed in the same sequential process, and (C3) the implementation programming language requires the programmer to specify execution flows, then program entanglement is inevitable. If two features do not interact, their programs do not have to entangle. In other words, feature interaction is the main reason for program entanglement. The entanglement conditions explain why existing programming languages cannot separate exception handling code from normal processing code. C1 and C2 are often dictated by the application, changing C3 is essential to solve the entanglement problem.

We call the condition under which two interacting features change their behavior their *interaction condition*. Presently, the programmer goes through code to determine when the interaction condition becomes true and *resolves* the interaction by changing code to specify the new behavior. The solution to N2 should enable the programmer to (R1) develop interacting features as separate and independent program modules, (R2) detect interaction condition among features automatically, and (R3) the features can be integrated and have their interaction resolved without requiring changing code.

We note that the language facilities C macros and Aspects separate code textually but do not meet these requirements. Empirical studies[3,4] conducted a decade apart have shown that Aspects do not significantly improve programmer productivity, despite studies that showed it can greatly reduce the lines of code to be written.

2.2 The verifiability challenge may not be solved by the verification tools alone

There have been significant advances in the art of verifying computer systems, especially in the model checking technology and in algorithms to determine the satisfiability of Boolean formulas. As a result, formal verification is becoming routine in hardware. But hardware designs are mainly composed of finite state machines and Boolean logic; in software a condition variable may be unbound and one must reason on predicates of complex data structure. Consequently, previous results in automatic software verification apply only to an *abstraction* of the software. The abstraction itself is a source of error and can rarely keep up with changes in the software.

More recently, a number of research groups developed model checking tools that apply directly to real programs. SLAM[5] is now a commercial product. CMC[6]

---

[3] G.. C. Murphy, et al, "Evaluating Emerging Software Development Technologies: Lessons Learned from Assessing Aspect-Oriented Programming," IEEE Trans. on Software Engineering, 1999, 15 (4).

[4] F. Filho, et al, "A Quantitative Study on the Aspectization of Exception Handling," Proceedings of ECOOP Workshop on Exception Handling in OO Systems, July, 2005.

[5] T. Ball et al, "The SLAM Project: Debugging System Software via Static Analysis," Proceedings of Principles of Programming Languages, 2002.

[6] M. Musuvathi et al, "CMC: A Pragmatic Approach to Model Checking Real Code," Usenix, OSDI, 2002.

reported the ability to handle software subsystems with tens of thousands of lines of code. But all of them also reported significant limitations.

The root cause of the limitation is the *state explosion problem*: the exponential increase in the state space that the model checker must explore as the number of state variables in a program and their value set increase. Model checkers for hardware benefits from very efficient Boolean SAT solvers. Existing SAT solvers for predicate formulas require iterations of solving NP hard problems. The effort spent by existing tools to compress the state space is already heroic (e.g. see[6]). To finally make this technology practical on software with scale will need additional innovation.

3 A language approach towards high confidence

3.1 A brief introduction to the Feature Language Extensions (FLX)

FLX is a set of programming language constructs that relaxes C3 and supports nonprocedural programming. A program unit in FLX consists of a condition part and a program body part. The program body part gets executed when its corresponding condition part becomes true; the programmer does not specify the execution flows of program units.

FLX provides language constructs for the programmer to organize program units into *features*. He develops a feature following a *model* instead of reading the code of other features. Features are integrated and have their interaction resolved in a *feature package* without requiring changing code. Interaction conditions among features are detected automatically. Thus FLX meets R1, R2 and R3. Features and feature packages are reusable and can be packaged differently to meet different user needs. FLX has been implemented on Java similar to the way that C++ added object oriented programming language constructs to C. Our experience of using it has been positive[2].

3.2 The verifiability of programs written in FLX

FLX is designed so that programs written in it will be amenable for automatic verification. An executable FLX program is compiled into a finite state machine even if its state variables are unbound. FLX provides language constructs for the programmer to provide input to support an efficient first order SAT solver to handle predicates of complex and arbitrary data types. This SAT solver is needed also for interaction condition detection. The FLX first order SAT solver is NP-complete as it first transforms a first order predicate formula to its disjunctive normal form. But it does not require iterations of solving NP hard problems as in existing algorithms.

4 Concluding Remarks

N1, N2 and N3 are necessary but not sufficient conditions. For example, it is not clear that one can *completely* verify a software system by assertions. But meeting the challenges will greatly improve confidence in software and programmer productivity. We just started the development of an operating system kernel subsystem using it, as well as a model checker for FLX taking advantage of the properties described in 3.2.

Bio: Dr Wu-Hon Francis Leung is an IEEE Fellow cited for his contributions to operating systems, protocols, and programming methods. He received his Ph.D. in CS from the University of California, Berkeley. He has more than 20 years' of industrial experience in software development and research on embedded systems at Bell Labs and Motorola. He joined the faculty of the Computer Science Department of IIT recently and can be reached via leung@iit.edu and (630) 697-4256.