DOI:10.1145/3201606

Harold "Bud" Lawson

Viewpoint
The March into the Black Hole of Complexity

Addressing the root causes of rapidly increasing software complexity.

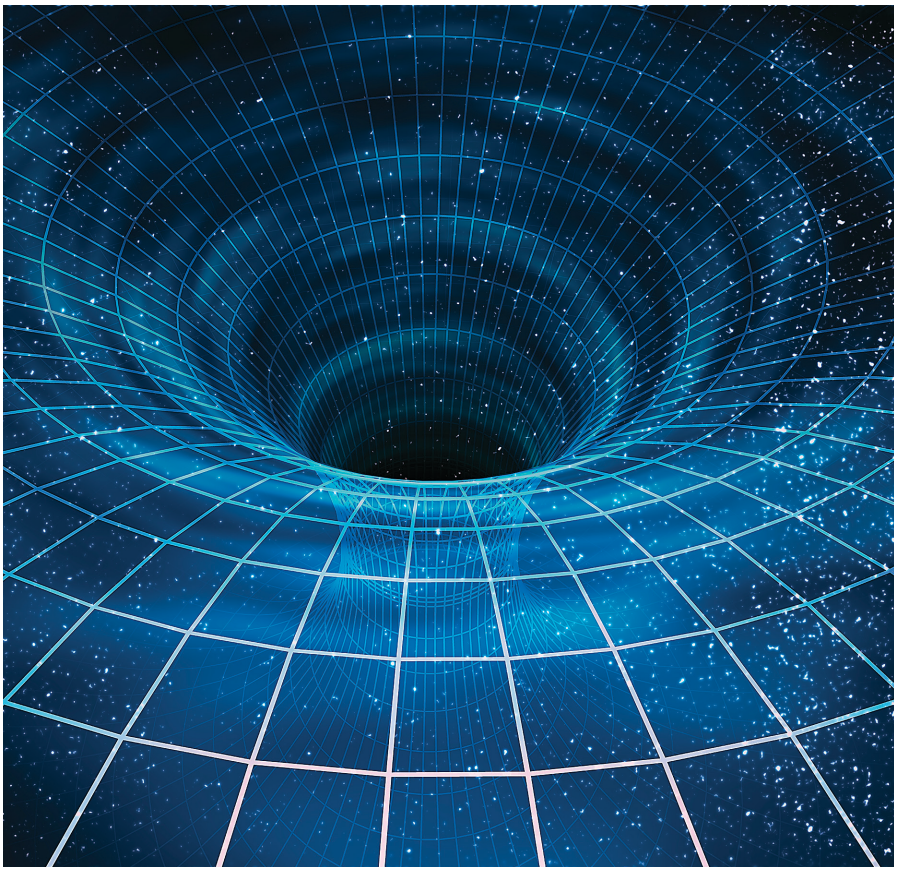

IMAGE FROM SHUTTERSTOCK.COM

In June 2002, *Communications* published my Viewpoint "Rebirth of the Computer Industry," in which I expressed hope that the past complexity sins of the computer industry had been admitted and that something positive could happen.[5] Instead, complexity has increased at an accelerated and alarming rate. I pointed to many fundamental problems in my previous Viewpoint; here, I emphasize two aspects: the hardware-software mismatch and the state of affairs in developing and sustaining software systems. Both are contributing root causes introducing enormous risks and impacting digital-age safety, security, and integrity. If the world ever hopes to climb out of the black hole of software complexity it will involve addressing these aspects in a constructive manner.

Hardware-Software Mismatch and Consequences

During the late 1950s, Robert (Bob) Barton pioneered the idea of designing the hardware to accommodate the needs of software; in particular programming languages in the design of the Burroughs B5000. For example, he used the Łukasiewicz's concept of Reverse Polish Notation as the means of expression evaluation. There were several other software relevant innovations as well. Burroughs continued to develop the B5500 and 6500.[1] I fondly remember Burroughs provided a pedagogical game that one could clearly learn the underlying principles for Algol and Cobol program translation and execution.

However, Burroughs did not dominate the computer industry. IBM did. With the announcement of the IBM System/360 series in 1964, the IT World entered a new era. Clearly, there were significant advances in the 360, in particular the usage of solid-state logic and the control of most models by microprograms. While the series provided compatibility over a range of models at various price/performance levels, there was, in my opinion, a very serious prob-

lem. The Instruction Set Architecture (ISA) included 143 instructions, but it was not easy to generate efficient code from the various compilers for Fortran, Cobol, and the new language PL/I.

The problems with providing 360 systems software have been well documented by Fred Brooks in his book *The Mythical Man Month*.[2] I was a member of the Change Control Board for OS/360 as representative for PL/I. So, I observed firsthand the rapidly increasing complexity. The original concepts for OS/360 were clear and described in a small notebook. Finding that the instruction set was difficult to deal with, compiler and other system software projects requested exceptions, for example in calling conventions and parameter passing. As a result the documentation volume grew rapidly and eventually it became virtually impossible to follow all the changes. Entropy was a fact; very few people read anymore. To add to the complexity, a very complicated Job Control Language was provided.

Given the situation IBM discovered a new market. Customers had extreme difficulty in installing and operating their System/360s. So, IBM established the role of "Systems Engineers" with the altruistic goal of helping customers making their installations and applications operational. The bug-laden software resulted in fix after fix, each fix reconciling some problems but introducing new ones. This all produced an enormous revenue source from the selling of Systems Engineer's services.

I identify this as the beginning of the March into the Black Hole of Complexity. It created fantastic opportunities for consultants and start-up companies that made fortunes because they could handle some piece of the complexity. Now very significantly amplified by a wide variety of suppliers of complex software systems.

There were efforts in 1960s and early 1970s to return to the importance of the hardware-software relationship. I led a research group at IBM that developed ideas of T-Machines and E-Machines supported by microprograms. T-Machines that implemented an instruction set conducive to constructing compilers and based upon ideas from Digitek (at that time a supplier of compilers). Compiled programs executed by E-Machines that implemented programming language-relevant instruction sets in a manner similar to the Burroughs computers.

There were several others following similar ideas including Wayne Wilner with the microprogrammed Burroughs B1700[10] and the work of Glenford Myers on software-oriented architectures.[8] I was involved in two new microprogrammable architectures where there were plans to not only emulate existing machines, but to use the T- and E-Machine approach. First was the MLP-900 at Standard Computer Corporation.[7] Unfortunately, due to a change in management, only the prototype was produced, but it wound up on the original ARPA network in a dual configuration with a PDP-10 at USC Information Science Institute. It provided a microprogram research facility and was used during the 1970s. Datasaab in Sweden planned to license the MLP-900 to emulate a previous machine, but also to work toward implementing T- and E-Machines. There, I designed the Flexible Central Processing Unit, a 64-bit machine, with rather advanced microprogram features. It was microprogrammed to provide a compatible D23 system, but the true potential of using the FCPU for T- and E-Machine implementation did not transpire.[6]

Had these innovative "language-directed architectures" achieved wide market acceptance one wonders how computing would be these days? But as previously stated, IBM dominated. In the mid-1970s, the microprocessor showed up and radically changed hardware economics affecting the product offerings by all computer hardware suppliers. While the integrated circuit technology was a major achievement the establishment of a primitive ISA (Instruction Set Architecture) that has permeated in the X86 architecture has had an even more radical effect upon the hardware-software relationship than the 360. Generating code for these processors is highly complex and results in enormous volumes of code. As a result, true higher-level languages have often been put aside and the use of lower-level languages like C and C++ permeate. Another trend evolved in attempting to hide the complexity, namely via middleware where functions provided by higher-level abstractions are translated often to C code. Clearly middleware hides complexity, but it is also clear that it does not eliminate it. Finding bugs in this complex of software levels is a real challenge.

The March into the Black Hole of Complexity created fantastic opportunities for consultants and start-up companies.

The problems continued when IBM in an effort to capture the personal computer market agreed to use DOS and other Microsoft software. This was followed by the so-called WINTEL cycle. More powerful processors with more memory from Intel—and then new software functionality (often not really essential and not used) from Microsoft and then the next round.

During the 1980s there was a debate about the merits of the CISC (Complex Instruction Set of the X86 type) versus RISC (Reduced Instruction Set) architectures. While RISC architectures provided enhanced performance and the fact that higher-level functions can be achieved by subroutines, they do not directly address the hardware-software relationship. That is, there is a "semantic gap" between true higher-level languages and the ISA. The semantic gap refers to the level of cohesion between the higher-level language and the ISA. The T- and E-Machine approach described here has significantly reduced the semantic gap.

We now know that the world has been provided with an enormous amount of new functionality via both CISC and RISC processors; but with the nasty side effects of the unnecessary complexity in the form of enor-

mous volumes of compiled code and middleware leading to bugs, viruses, hacker attacks, and so forth, to a large extent due to the enormous complexity. Given the current problems of cyber security, it is high time to seriously address the semantic gap and develop language-directed architectures that make software more understandable, maintainable, and protectable while also significantly reducing the amount of generated code.

Software Development, Deployment, and Sustainment

I have termed the unnecessary complexity that has evolved Busyware. Certainly it keeps vast numbers of consultants and teams of software engineers and programmers occupied. They should be focusing on producing good-quality software products (Valueware and Stableware), but find themselves often side-tracked into handling implementation complexities. While a resolution of the hardware-software mismatch would be a step in a positive direction, the scope and complexity of today's large software endeavors demand that the process of developing, deploying, and sustaining software must be improved. For example, today's operating systems and many advanced applications in the range of 50 to 100 million lines of code produced by a large group of software engineers and programmers have introduced significant problems of stability and maintainability.

The software engineering profession was established to improve capabilities in developing, deploying, and sustaining software. While early efforts focused on improvements in program structure, later developments have focused on the way of working. Many "gurus" have provided their own twist on best practices and methods that are often followed in a religious manner. As a result, a plethora of approaches have evolved. While this situation has made a lot of people rich, it certainly has contributed to additional complexities in selecting and applying appropriate practices and methods—moving us even deeper into the Black Hole of Complexity.

In an effort to improve upon this alarming situation an international effort initiated by Richard Soley, Bertrand Meyer, and Ivar Jacobson resulted in the SEMAT (Software Engineering Method and Theory) organization. They observed that software engineering suffers from:

- The prevalence of fads more typical of a fashion industry than of an engineering discipline;
- The lack of a sound, widely accepted theoretical basis;
- The huge number of methods and method variants, with differences little understood and artificially magnified;
- The lack of credible experimental evaluation and validation; and
- The split between industry practice and academic research.

The scope and complexity of today's large software endeavors demand that the process of developing, deploying, and sustaining software must be improved.

As a concrete step a team of international experts developed the Essence Kernel[4] that has become an OMG standard.[9] Essence provides:

- A thinking framework for teams to reason about the progress they are making and the health of their endeavors.
- A framework for teams to assemble and continuously improve their way of working.
- The common ground for improved communication, standardized measurement, and the sharing of best practices.
- A foundation for accessible, interoperable method and practice definitions.
- And most importantly, a way to help teams understand where they are, and what they should do next.

While developed for software engineering, when examining Essence it is obvious there are many ideas that can be applied on a wider scale. Certainly as we move into the era of the Internet of Things and cyber-physical systems the importance of organizing multidisciplinary teams and systems engineering become obvious. To meet this need, Ivar Jacobson and I have co-edited the book *Software Engineering in the Systems Context* where many well-known software and systems experts have provided their input.[3] Further, a call is made to extend the ideas from Essence to the systems engineering domain.

Conclusion

We are very deep in the Black Hole of Complexity and two important root causes to this situation have been identified. First, due to the mismatch between hardware and software that has resulted in complexities with the widescale usage of lower-level languages and middleware. Secondly, given the scope and complexity of today's software systems, we need to improve our approach to developing, deploying, and sustaining software systems that have become the most important and vulnerable elements of modern-day systems. Here, Essence provides an important step forward. Progress must be made in these two important aspects if the world is to avoid sinking further into the Black Hole of Complexity. C

References

1. Barton, R. Functional design of computers. *Commun. ACM 4*, 9 (Sept. 1961).
2. Brooks, F. *The Mythical Man-Month*. Addison-Wesley, 1974.
3. Jacobson, I. and Lawson, H., Eds. *Software Engineering in the Systems Context, Volume 7*. Systems Series, College Publications, Kings College, U.K., 2015.
4. Jacobson, I. et al. *The Essence of Software Engineering: Applying the SEMAT Kernel*. Addison-Wesley, 2013.
5. Lawson, H. Rebirth of the computer industry. *Commun. ACM 45*, 6 (June 2002).
6. Lawson, H. and Magnhagen, B. Advantages of structured hardware. In *Proceedings of the 2nd Annual International Symposium on Computer Architecture*, 1975.
7. Lawson, H. and Smith B. Functional characteristics of a multi-lingual processor. *IEEE Transactions on Computers C-20*, 7 (July 1971).
8. Myers, G. SWARD—A software-oriented architecture. In *Proceedings of the International Workshop on High-Level Language Computer Architecture*, 1980.
9. OMG. *Essence—Kernel and Language for Software Engineering Methods*. Object Management Group, 2015.
10. Wilner, W. Design of the Burroughs B1700. In *Proceedings of the Fall Joint Computer Conference*, 1972.

Harold "Bud" Lawson (bud@lawson.se) is an ACM, IEEE, and INCOSE Fellow, IEEE Charles Babbage Computer Pioneer, and INCOSE Systems Engineering Pioneer.