Response to Comments on
A Foundation for the DoD Digital Transformation

Walter W. Wilson
Lockheed Martin
March 13, 2021

My project proposal, "A Foundation for the DoD Digital Transformation", has been officially rejected. But some comments and criticisms were made that deserve a response. I think I can answer those criticisms and make a stronger case for my project, but it involves more speculative arguments that I usually avoid. Section A gives a more detailed discussion of the potential benefits of axiomatic language. Section B discusses the grand challenge of axiomatic language implementation. Section C talks more about the application of the language to CAD.

A. Benefits of Axiomatic Language

1. Specifications Should be Smaller

A common assumption in software engineering is that programming time per line of code is roughly constant, regardless of language level, and this is supported by data [Prechelt 2000 "An empirical comparison ..."]. So a possible programming productivity benefit for axiomatic language resolves to the question of how much smaller would axiomatic language specifications be than the equivalent programs in other languages. Axiomatic language is intended as a pure specification language where one just defines the external behavior of a program – inputs/outputs as seen by an external observer – without having anything to say about the internal processing. Automatically generating an efficient program for this specification is the task of the language implementation, described in section B. A program in a conventional programming language defines the internal processing of the computer and implicitly defines the external behavior from the input/output operations. But that means the axiomatic language specification information is a proper subset of the internal processing information in the conventional language program, and thus should be smaller. (The missing implementation information for the axiomatic language specification would be supplied by the transformation system of Section B.)

How much smaller would a pure specification of just external behavior be? For a program that is mainly input/output, it might not be much smaller at all. But for other programs the reduced information size may be significant. Let's say that the pure specification information would

typically be half the size of implementation information and thus an axiomatic language specification should be on average half the size of a conventional program for the same problem. Thus, by the above programming productivity lines-of-code rule, there should be a doubling of productivity using axiomatic language. And this benefit would apply across the entire software industry. How many billions is that?

2. Greater Reusability

In addition to the benefit from smaller specifications, axiomatic language may provide increased programmer productivity due to greater software reuse. A minimal language like axiomatic language will have a standard library of general-purpose utility functions which application programmers will use instead of having to write the utilities themselves. The lines-of-code for these utility functions would not be counted as part of the application code, since ideally they would be used for many other applications. Thus, the application programmer's productivity goes up the more that application code can be replaced with use of general-purpose code from a library.

My impression of axiomatic language is that it would provide greater reusability than conventional languages for the following reasons:

(1) Higher-order forms can be defined in axiomatic language that allow one to abstract general-purpose patterns from similar application code. This can be difficult to do in conventional languages.

(2) The metalanguage capability of axiomatic language allows one to define language abstractions that hide the details of an application domain and this could reduce application lines of code. (See "language-oriented programming".)

(3) There is often a lot of boilerplate code with defining new functions in conventional languages and this is largely absent from axiomatic language.

(4) Axiomatic language supports "fine-grained modularity". That is, any utility function definition that saves a line or two of code will probably provide a payback.

(5) There may be an execution efficiency penalty in using a general-purpose algorithm for a specific purpose in a conventional language. In axiomatic language, with a good implementation transformation system, there should be little penalty since one is just re-using specification "definitions".

How much can greater reusability reduce the size of application code? This will have to be shown by examples.

B. Implementation of Axiomatic Language

Implementing axiomatic language means automatically transforming specifications to efficient programs -- a grand challenge of computer science. The system would have to "understand" the user's input specification and then, from that understanding, generate an equivalent efficient program. Such a system would need comprehensive built-in knowledge of programming concepts like utility functions, data types and their representations, specification patterns like higher-order forms, implementation algorithms, etc. This would be an encoding of the same knowledge that a human programmer has. The knowledge for implementation algorithms would include pre-stored proofs to guarantee equivalence of the generated program with the input specification. (The system would not be expected to invent algorithms or discover proofs at transformation time.)

An initial, proof-of-concept transformation system for axiomatic language would have some limited encoded programming knowledge sufficient to transform some tiny examples. That is the main task of this project. Transforming additional examples would require adding additional knowledge. A key problem is to come up with an efficient algorithm that can match an input specification against a potentially large knowledge base of programming concepts – a milestone of this project. Another milestone of the project is to incorporate a proof of equivalence into at least one of the transformation examples.

Evolving the initial proof-of-concept transformation system into one suitable for production use would be a long process of incrementally adding programming knowledge. New specifications for new problems will typically require new knowledge to be added, much like a human programmer would have to be taught new programming concepts. Over time, as the knowledge base grows, adding new knowledge should become less frequent. Eventually the system should reach a point where many new specifications can be transformed automatically.

Is completely automatic transformation of arbitrary specifications even possible? One can argue from computability that the answer is no – no system can successfully automatically transform all possible specifications that a user might write. An argument could go like this: If a specification produces no output, then the efficient program for that specification would be one that immediately halts. But it is undecidable whether or not a program (or its specification) produces any output, so such a transformation system (which effectively would solve this undecidable problem) cannot exist. But we probably don't need a system that can automatically transform all possible specifications. Instead I would consider an axiomatic language implementation to be a success if it can *automatically transform straightforward specifications for most typical problems*. If an input specification is not so straightforward or the problem not so typical then an expert would need to add knowledge to the transformation system so that automatic transformation can proceed.

How big would a knowledge base need to be for a transformation system to be "successful"? Probably a comprehensive knowledge base would need many thousands of programming concepts – but probably not millions. At some point new concepts will often be seen to be variations or

instances of earlier concepts. Defining this knowledge along with proofs and test cases may require thousands of programmer-years of effort. (Perhaps on the order of Linux.)

How could we develop such a large system? Who would pay for it? I think Wikipedia shows the way. The 6,000,000+ English articles is a remarkable compilation of human knowledge, created largely through "volunteer" effort. (Some organizations may sanction their employees to do Wikipedia edits.) A similar system for representing transformation programming knowledge would allow the incremental addition and refinement of this knowledge, done by "volunteers", and driven by user needs – but with much more careful editing. The huge development cost wouldn't show up as a big part of anyone's funding budget. But even if one estimates the real cost of the "volunteer" labor, the true total implementation cost may be justified by the potential software engineering benefits.

C. Application of Axiomatic Language to Computer-Aided Design

The definition of a CAD system in axiomatic language would be a huge programming endeavor. It would require an exceptionally capable transformation system – far beyond the results of this project. But I would argue that the advantages over commercial CAD could be significant:

(1) Textual, human-readable design data instead of a CAD vendor's secret, proprietary, binary file format – A textual definition makes our design data more accessible. It is no longer held hostage by the CAD vendor. The data would be more self-documenting. Additional comments could explain *why* design decisions were made. This information could be useful to future engineers and may not always be captured when designing with interactive CAD systems.

(2) High-level scripting language for design automation and optimization – There are two approaches to computer-aided design: (1) interactivity and (2) batch processing. Interactivity is nice, but batch scripts (programs) can be a more powerful way to define geometry. Scripts capture engineering knowledge and automate processes. They record design history and "intent". They provide repeatability, which could be needed for future design modification. Boeing does a lot of scripting using their amazing GEODUCK system, which can optimize the shape of an entire airplane. [Getting Math Off the Ground: Applied Mathematics at Boeing , Grandine 2014 “The Case for Scripted Process Design and Engineering”, Grandine - PNW Numerical Analysis Seminar 2015] Boeing scripts encode their entire "design guide" – 1000s of pages. Some engineers still prefer interactivity, but others have taken to writing short Python scripts to formulate and solve their design problems. "They drool over it." It should be mentioned that there are systems like sketch-n-sketch that combine interactivity and scripting.

(3) Open-source geometric engine – Our CAD system should not be a black box! The mathematical details of design data representation and geometric algorithms should, in principle, be accessible to the engineers that use them. A problem may arise that requires inspection and

tweaking of a mathematical algorithm. A new application may need low-level customizations of the mathematics.

(4) Language extensibility supports the definition of new design features – The extensibility and metalanguage capability of axiomatic language, along with having an open-source geometric engine, would better support the definition of new design features. Features like holes and fillets are convenient abstractions for defining geometry. For aerospace projects, higher-level features like area rule and wing twist may be useful. Introducing new design features using a CAD system's API can be painful, but having complete source-code access to a geometric engine should make such customizations easier.

(5) Language extensibility supports the definition of new surface types – The STEP standard and some commercial CAD systems have locked our geometry in 40-year-old trimmed-NURBS technology. New surface types on the horizon can provide desirable attributes that go beyond what is possible with trimmed NURBS: local refinement, watertight geometry, linking analysis and design, support for volumetric modeling, support for 3D printing and new composites, etc. An open-source geometric engine would be helpful, of course, to define new surface types. And we wouldn't have to wait for some future version of CAD system.

(6) Exact symbolic definitions of geometry – The symbolic nature of axiomatic language allows a scripting language to capture exact symbolic definitions of geometry. If a curve on a surface is a geodesic between two points, we should record that in the definition. Similarly, symbolic constants like cos30deg or decimal constants like 0.1 should be saved in their original form instead of their binary approximations.

(7) Proof of correctness may be possible – Design automation carries risk. GEODUCK reportedly can optimize the shape of an aircraft structural part. But if there is a bug in that optimization, the part could be too weak – an airliner could crash! There may be a future requirement to prove that CAD software produces safe airplanes. If Lockheed Martin gets back into the commercial aircraft field (https://en.wikipedia.org/wiki/Lockheed_Martin_X-59_QueSST), it could be our requirement. Floating point arithmetic makes proof of correctness difficult for numerical programs. A floating point function y=f(x) is only defined for floating point values x – discrete points on the real number line. The derivative may be positive (y'=f'(x) > 0), but there may be adjacent floating point values x1,x2 where f is decreasing (f(x1) > f(x2)). This type of logical inconsistency may cause a problem for a correctness proof. Axiomatic language has no built-in arithmetic so one would have to symbolically define some form of approximate arithmetic. But carefully-defined approximate arithmetic, combined with exact geometry definitions, may help one avoid logical inconsistencies and make correctness proofs possible. And compared with other languages, axiomatic language is particularly well-suited to proof: http://axiomaticlanguage.org/proof.htm . Proof is the future of software!

(8) Better support for long-term design data preservation – Design data needs to be preserved for the lifetime of an aircraft program, maybe a century for some airplanes. With commercial CAD systems, this can require costly migration from one version of the CAD system to another. For long-term sustainment we will want the ability to reanalyze and redesign parts as needed, which means we need to be able to rerun the CAD system and all our design and analysis scripts and support software. (I recall an Air Force requirement that we must preserve all data and software that "touches" a design for up to 8 years after the aircraft program ends.) The STEP standard is not much help here since it doesn't save all the high-level design features of a modern CAD system and it doesn't save our customization applications. When rerunning CAD software to redesign a part, we would want the results to be reproducible. To achieve this for numerical geometric algorithms, we would need to preserve the geometric engine. We would need to preserve the geometric engine and other software in source code form to help ensure that it could be rerun on future computers. In fact, I would argue that we need reproducibility down to the last bit! If a design script does a point-in-solid test and the point happens to lie approximately on the solid's surface, then the tiniest numerical difference could produce a different test result and affect the final design. Since future computers may have differences in floating point numbers, we will need explicitly-defined approximate arithmetic to guarantee identical results far into the future. Archiving the CAD source code along with the design data means that the programming language for the CAD software would be the standard for long-term preservation. The small size and elegance of axiomatic language would make it an ideal long-term standard – orders of magnitude smaller than STEP. (See Thoughts on LOTAR and GD19_slides.pdf)

(9) Commercial CAD pricing vulnerability – CAD vendors effectively hold our design data hostage. If they were to raise their prices by, say, a factor of 10, we would probably have to pay it. Is this situation acceptable? My solution is that we should move to an open source geometry representation where our data is not held hostage by a CAD vendor. (See A_Vision_for_CAD_released.pdf.)

(10) May be cheaper! – An open-source CAD system (and the axiomatic language implementation) may cost less than what the DoD indirectly spends on commercial CAD licenses, especially considering many decades of sustainment. And a non-commercial CAD system may help us avoid costly CAD migration.