ProgXML

Programming-XML

Concept Paper / “Light Green Paper”

Informal Peer Review Copy

http://www.mhknowles.net/Papers/ProgXML-IPRC-mhk.pdf

Michael Hugh Knowles

June 21, 2011

Advisory

When XML first came out, I named the concept I am trying to promote in this paper "ProXML", short for "Programming-XML". Well, over the years, other people started using "ProXML" for various and quite different things, and some have made trademark claims on it. So, I decided (in early 2010) that I would change my term to "ProgXML", which no one else seems to have laid claim to or even used. I consider "ProgXML" to be a generic term, and I intend that it and any terms sufficiently generically equivalent be in the public domain. But I am also intending that the term in the public domain refer to the concept outlined in this concept paper

Abstract

This concept paper—a "light green paper", a variant of the term more common in the European community—proposes that an extension of XML be researched & developed so as to eventually become a replacement for standard High-Level Languages for *standard* computer program development, eventually entirely replacing today’s standard general HLLs, and allow easy extension to any special purpose kinds. A brief history is given, with a quick description of some of the first baby steps already taken along these lines, such as MathML and Microsoft’s WPF and XAML. Two “laundry lists” give comparisons between standard HLLs and a possible Programming-XML/ProgXML approach in terms of their respective qualities, limitations, and future potentials. (It is the intent of the author that the term “ProgXML” and other sufficiently generic terms be in the public domain.)

NB: ProgXML is a completely different concept from what have become the usual “XML-based programming languages”, which are more-or-less standard High Level Languages, except that XML is used to implement them. They have only their single, standard HLL mode for program entry-editing-presentation/viewing.

KEYWORDS

Programming Languages, Programming Techniques, Programming Development Systems, Programming-XML, ProgXML

Introduction

Software has made great strides in some very important directions in the last half century. Managing data, previously paper based, has become much easier with database developments, especially since relational databases evolved in the ’60s, and capable of more “intelligence” and extensions thereto since the introduction of XML.

User interfaces to computers, which transcend many of paper’s limitations and have even started to replace it, have come a long way since the binary switches and punch card decks of the ’40s and ’50s (actually, I used both in the ’60s and even the ’70s), and have started becoming more intelligent and easily extensible since e.g. we started handling forms with XML. Today’s user interfaces are *beginning* to be heavily graphically oriented (so far limited to 2 or “3” dimensions), a feature which greatly extends—almost “transcends”—the inherently “linear” nature of the older pre-graphics user interfaces. (E.g. “printers” used to be “line printers”.)

And, with the evolution of “word processing”, which evolved from a glorified marketing gimmick to sell typewriters-now-crudely-interfaced-to-computers to a still evolving reality, visually sophisticated paper documents are much easier than ever to develop and manage in ever more dismaying quantities, with an ever increasing market share being picked up by "totally" electronic production and distribution, together with computer-based *intelligent* search and management possibilities (and again we see databases and user interfaces) since the introduction of… XML again. We are also seeing the growing popularity of "e-documents", which can update e.g. spreadsheet data in "real-time" (even if frustratingly slowly at times, and at the mercy of the vagaries of one's wideband service) and can even be interactive with the user (more on which later).

XML – eXtensible Markup Language

XML ("ben SGML") has become a universal darling in the decade since its development in 1996, followed two years later by the release of XML 1.0 by the World Wide Web Consortium (W3C). **XML** (e**X**tensible **M**arkup **L**anguage) was designed to be an Internet-oriented version of the more purely document-oriented SGML (Standard Generalized Markup Language). XML is explicitly semantics-based, as opposed to "text/presentation-based" where the semantics is hard-programmed into the compiler that "decodes" and "compiles" the text by means of the implicit "syntax" of said text. Uses are now commonly seen for XML in all directions, all directions except for the *whole* computer programming and software development process itself. This article hopes to inspire the eventual alleviation of this oversight.

The Early Days of Computer Programming

Computer programming started as instructions and data entered through physical switches as machine-level binary (as opposed to higher-level codes to be decoded later). The first *big* advance over this was "assembly language", where the instructions and data were entered as text/character-based mnemonic codes/symbols and number representations. Punch paper tape and punch cards were adapted for external storage of computer data and programs, and of course for loading programs and data into the computer through an appropriate electro-mechanical interface. Scalability of assembly language programs was found to be a problem early on, although multi-million line programs were written "successfully", with maintenance headaches scaling up quite a bit faster than the program size.

The next—and last—big evolutionary advances (as opposed to "merely" important advances) in computer programming/software development came with the evolution of High-Level Languages (HLLs, such as FORTRAN, LISP, ALGOL, COBOL, C, C++, JAVA, C#), the storage of programs and data on computer accessible electro-mechanical devices (drums, disks, and magnetic tape drives) in files in a "high-level" file system (with naming, etc.), and moving the user interface from punch cards to electronic terminals linked directly to the computer. The latter was a lot like punching cards and putting them through the card reader, except that first entry and later editing was through a character based (non-graphics) terminal, and the files resided on the disk (and hopefully on backup tape, in case of all too frequent accidents). Graphics terminals are now common, but programs are still stored as "lines" of text (of e.g. ASCII characters) representing the computer program code in a file on a disk, i.e. glorified electronic punch cards.

The scalability problems associated with assembly language (across the board: design, programming, debugging, maintenance, etc) were perhaps the major driving factors in the switch to HLLs, not just for applications but for systems programming. Our HLLs have continued to evolve considerably beyond FORTRAN and COBOL, the first widely used HLLs, first introduced in 1957 and 1957, respectively, but they are still one-dimensional representations of the vastly higher number of logical-linguistic-conceptual-ideal dimensions that today's HLLs and programs attempt to deal with. (LISP, first introduced in 1958, never gained wide popularity, except—very importantly—in the early world of Artificial Intelligence—AI.) These HLL-based programs are still held in perhaps the simplest of today's databases, these collections of one dimensional, lines-of-text-oriented files, in hierarchical file structures, much as word-processing document files once were, and still are for that matter.

Although interlinking can become complex, the HLL text in these files is piece-wise linearly input into a very large and complex program, the compiler (for that HLL). This process may leave either a fully-linked executable, or one or more pieces of partially constructed program that need to be linked, perhaps dynamically, before/while the program is later executed. The compiler, through a very complex grammar-oriented process, "lexically analyzes" then "parses" or "syntactically analyzes" the text—i.e. "recognizes" the program's lexical entities implicit in the one-dimensional text, then "recognizes" the syntactic-grammatical constructions implicit in the linear arrangement of those lexical entities—and emits the code to perform the semantics implicit in the syntactic-grammatical constructions. (The higher the level of the language, the more "implicit" the syntax and associated semantics are.) Multiple HLLs are allowed in more sophisticated systems such as Microsoft's .NET framework.

Scalability problems still abound. HLLs postponed them, but did not eliminate them. Large programs are still unwieldy to design, program, debug and maintain. A variant of Parkinson's Law—or Malthus—seems to hold in programming: people always want to create programs that outstrip resource limitations, with scalability limitations of the HLL-based programming process itself tending to lead the pack.

Microsoft's .NET Framework(s)

Microsoft describes its .NET as (paraquoting) "a set of software technologies for connecting people, systems, information, and devices that is built on a foundation of XML Web services, so that new and existing applications can connect with software and services across operating systems, programs, and programming languages." These "operating systems" and "programs" are all programmed in the standard HLLs or assembly languages referred to as "[across…] programming languages", even if some of the interfaces, modules or tiers are "programmed" in an XML-based implementation. Examples are Microsoft's Windows Presentation Foundation (WPF), which is a graphical subsystem of .NET Framework 3.x (formerly called WinFX), and Microsoft's eXtensible Application Markup Language (XAML) used in WPF to define objects and their properties, relationships and interactions. The .NET framework does not, however, use any ProgXML type approach for the explicitly "traditional programming" part of the software development processes/programming activities involved, only for the implicit part inherent in the XML databases that facilitate the interfaces and interconnections (which, of course, are an essential part of the "software technologies", and do need to be "programmed").

Early Bootstrapping

Back in the early days of computing, programmers used assembly language to program primitive HLL compilers which were then used to program more sophisticated compilers in a bootstrapping process that eventually yielded an almost pure HLL approach to both systems and applications programming. Although HLLs are used to program the XML systems generally used today in more highly intelligent interactive databases and electronic documents, our most common software development methods and methodologies have not subsequently utilized XML to bootstrap XML-based software development/programming technologies. The situation is ripe for a new wave of the future of software development: "**Programming-XML**" or "**ProgXML**", of which we will offer here a "Pie-in-the-Sky Portent".

ProgXML

PRE-REACTION WARNING: the reader may find self saying to self something like "This isn't new! I've heard of people using XML this way." This is just the point that needs and wants to be made here. In areas where there has been no pre-existing HLL, and insufficient desire on the part of management to spend

$$$ to develop and maintain yet another HLL-compiler-etc complex, people have tried an XML-based approach for $ or $$ instead of $$$ or even $$$$. They have just *never* tried a seriously XML-based approach to *replace* our current day-to-day workhorse HLLs—C#, JAVA, C++, FORTRAN, COBOL (you *should* laugh, but remember this: I know a guy who makes $400K a year converting old COBOL into "new" COBOL).

"**ProgXML**": Computer programming itself needs to start moving in the direction of one of computer science's greatest advances: **XML**. We need to start evolving a programming version of XML to do software development with ProgXML-based program-data-base software interactively linking the applications/systems and their developers/managers (human or machine) to the programs under construction/debugging/review/-upgrading, and further to "distributed" co-developers/-managers, user libraries, debugging tools, project management tools, documentation, tutorials, databases and other resources the program and/or developers/programmers will use, etc. The possibilities for optimizing software development with regard to all the usual performance metrics, including speed, security, and R&D costs, are immense.

“ProgXML” should not be thought of as a yet another High-Level Language, nor as a family of HLLs. In particular, it is not like LISP, once thought of as the HLL answer to the intractable problems of Artificial Intelligence, or even FORTH, which is actually a Multi-Level Language (which had great potential for evolution until it was prematurely standardized, an all too common death-knell for ideas with great potential, even if “their time has come”).

The idea is that the programmer will interact with the ProgXML system to describe, “internalize” and manipulate the semantics (and pragmatics) of the desired “program” or “subprogram” software, but not—necessarily—with a standard HLL-type syntax for doing this. In fact, multi-dimensional modalities are sure to evolve quickly, bootstrapping from what are now standard GUIs.

It will be somewhat easier if we digress and just think in terms of presenting-rendering or displaying existing code (“legacy code”). The software code—i.e. its semantics *and* “pragmatics” (e.g. errors relating to hardware limitations in a target computing environment)—would exist independently of *any* HLL. Just as XML allows many different ways of presenting-rendering-displaying the data held in an XML database, ProgXML would allow—not as a primary modality, except perhaps early in the bootstrap evolution of ProgXML—the program or subprogram to be displayed as (what used to be called) “source code” in C# or JAVA, or in any other HLL, as long as the ProgXML software has been itself extended to allow presentation in that HLL. Eventually the presentation or display of existing code would primarily be in whatever interactive modalities are evolved for use with ProgXML, and the extensibility of XML means that new presentation modalities can always be programmed to present/display (for viewing or modification) new code as “legacy code”, i.e. as pre-ProgXML HLL “source code” with correct syntax, along with other indications of its semantics and pragmatics. It could even generate compilable HLL source code, for legacy target situations.

Now let’s switch back to entering and manipulating program semantics (and—we should emphasize—pragmatics). Initially, i.e. early in the evolution of ProgXML, the programmer could input/enter program semantics e.g. in C# and see its presentation in JAVA. S/he could use different HLLs, according to the appropriateness of that HLL for viewing or modifying the particular bit of code. Or multiple programmers could each be programming in whatever HLL they chose and viewing the presentation of someone else's existing code again in whatever HLL they chose.

New programming interfaces would quickly be evolved to take advantage of ProgXML's extensibility. For example, we use 2 dimensions to interface with spreadsheet, and we could just as easily start to use 2 or "3" or more dimensions to input program semantics (for e.g. complex decision tables, which are almost impossible to program correctly using standard "structured programming", one of our "legacies" from the '70s). We use forms to enter data in databases, where the data is checked for validity, and we can just as easily use forms to enter program semantics, with immediate syntax validity checking, even if not 100% effective error detection. The forms would have built in "help" and "wizards" to aid in code semantics and pragmatics input. Since some code semantics has already been entered, at least some of the newly entered semantics can be checked for semantic validity, e.g. the proper use of an already (well-) defined variable as a subroutine parameter, not just by syntactic data *type*, but by semantic data type and particularized data attributes. The same goes for checking pragmatics, e.g. if you know in advance that you want to run the program on both 32 and 64 bit processors.

MathML as an Example

MathML, for example, which already allows math *calculations* to be done (somewhat) portably in Mathematica-Maple-MatLab-MathCAD type environments, is already being developed somewhat along these lines for use in electronic documents and web pages, which will then be capable of interactively producing calculations, graphs, plots and so on for the reader, within limits that can be set by the provider. E.g. a user might be able to choose from a list of (or describe arbitrary new) wavelets, place and orient them, and then watch a movie of them interacting with/filtering/transforming a system with user selectable parameters. XML has already found a permanent home in popular spreadsheet software. This wave of the future for interactive mathematics can—and must—be extended to software development.

ProgXML Evolution

Programming-XML software will go much further and give developers the software R&D version of WYSIWYG (which we can here consecrate as "WYPIWYG"—"What You Play-with Is What You Get"), but oriented toward the dynamics of instantaneous testing, distributed programming, program distribution (including "Windows Live" type products) and execution, which must all go way beyond the statics and dynamics usually associated with the XML-based combinations of electronic&paper documents that are now evolving.

Ease of use can be expected to evolve extremely rapidly. For example, as already hinted at above, syntax will no longer be as problematic as now (compile; get error message; "decode" error message; re-read manual; put right punctuation in right place; re-compile; repeat indefinitely) since at worst the researcher-developer will fill in a form—that insists on completely correct "syntax", and possibly on correct semantics and pragmatics, within limits—with labeled and on-screen documented text fields for the elements of e.g. an object declaration or a heavily parameterized procedure call to a library/API routine. This will catch *all* syntax errors, and many semantics errors, and even some pragmatics errors. Immediate extended feedback/-debugging at entry-time of *some* program semantics—internally held as statements/expressions/etc in a ProgXML "semantics-pragmatics markup language"—likewise becomes almost trivial to offer. Programs can be viewed and interacted with in various dynamically developer-chosen "presentation markup renderings", initially in various popular HLLs (which will be important at first), and eventually in highly evolved ProgXML-GUI-based presentation-display styles.

Not only will standard programming practice be optimized, but the newer software development styles of "agile software development" and "extreme programming" will benefit greatly from a ProgXML approach. The benefits of their strong points should be easy to improve significantly, and their weak points should be much easier to overcome using ProgXML.

ProgXML Extensibility

Except for user-named subroutines, standard HLLs only allow explicit statement of that particular HLL's linguistic-conceptual constructs. The "high-level meaning" of, for example, an assignment statement will be implicit, and often difficult to discern. ProgXML will allow the developer to explicitly eXtend the semantics "Markup Language" and its concepts/constructs to match the desired program much like the famous Dutch computer scientist Edsger Wybe Dijkstra envisioned in the late '60s, and to likewise keep the semantics of *all* levels of the program not only explicit, *but computer accessible semantically* in a way not achievable with standard HLL inline comment text.

Digression: FORTH is the only language currently extant that comes close to fulfilling Dijkstra's dream of a "structured programming" system in which one would explicitly design "languages" (expressed in those days in the structuring and naming of subroutines, variables and other data structures) to solve the *classes* of problems the programmer faced/perceived. These classes of problems and the corresponding languages to solve them needed to be designed in various levels 1) to implement in lower levels the concepts/constructs of the higher levels, and then 2) to express the problem(s) solution(s) at each level, including the "highest level". Unfortunately, FORTH lost its huge *potential* for popularity when average program-size ballooned and scalability became all-important. Also unfortunately, *Dijkstra's* concept of "structured programming" (as first designing languages to solve *classes* of problems and then to express in those languages *particular* solutions to the *particular* problems in those classes that were perceived to be “the ones”, evolving those solutions to match the evolving problem perceptions) lost out as a programming paradigm-methodology, at least in the US, to the wave of “top-down structured programming” paradigm-methodologies that swept the US. These latter merely structured *particular* solutions to the perceived problems, without the insights that came from in-depth analysis of the problems as being exemplars of a class, and even analysis of the “perceptions” themselves. “Top-down structured programming” was/is a far less dynamic and “malleable” approach than Dijkstra’s… “might have been”. **End Digression.**

This article is intended to spark interest in and promote the initial evolution of ProgXML-based software development. ProgXML systems will allow much more sophisticated software packages of any size to be developed much more rapidly than at present. This Programming-XML concept can be compared with the Internet in importance, although it will directly affect an entirely different community. It certainly will not—initially—be as universally visible to the average computer user as the Internet. On the other hand, it will forever change the world of software development.

A Quick Laundry List Comparison: HLL(s) versus ProgXML(s)

HLL characteristics currently include:

1. scalability problems in all aspects of HLL-based program development are already perhaps the greatest limiting factor for developing large sophisticated programs; the semantics that HLLs can efficiently express and implement only extend through a narrow band of the dynamic range of the problem semantics;
2. inherently one-dimensional text/implicit-semantics-based as opposed to ProgXML-style potentially multi-dimensional (presentation and manipulation interfaces) explicit-semantics-based; *text* implicitly retains the "values" (in many senses) and "meanings" of the program in the *syntax* of the expressions in the HLL; code reuse is often cost-ineffective because reuse is a *semantic* operation, and syntax is an inefficient means and not an end;

3. error-prone text editing, with syntax/semantics error detection delayed until (psycho-ergonomically non-optimal) compile/build/run/-debugging time; truly intelligent, truly program-aware text editors infeasible to develop; e.g. text-based but no explicit semantics-based search/replace;
4. syntax so complex that HLLs are literally as difficult and time consuming to learn as foreign languages like French or German;
5. languages/compilers very difficult to extend;
6. text is input to lexical and syntax analyzers so complex that language/compiler designers often specialize over their entire lifetimes; (paraphrasing the old joke: “it takes their entire work-lifetime to learn to do what they should be doing their entire work-lifetime”;)
7. the “presentation markup” (i.e. the listing of the program text in e.g. C#) IS the “semantic markup”; no flexibility in “presenting” program semantics to the developer or manager;
8. large amounts of computer time dedicated to (quasi-) redundantly re-compiling the massive text-data-based source files (usually *many many*

times, often just to eliminate a few syntax errors in a single program statement; and many more times to debug semantics);

9. incremental compiling and linking/building not the simplest procedures;
10. "non-malleable" programs ever more difficult to maintain as they "mature", reaching old-age and even senility all too early in their life-cycles;
11. The more complex the objects in Object Oriented Programming (OOPS), the more tedious and difficult their input becomes with standard HLLs;
12. programs with limited portability (for many more reasons than mentioned above);
13. many languages, with many dialects each, most of which can be made compatible only at significant cost, further degrading portability and/or reuse of established program code; i.e. difficult to "mix-and-match" languages and the code already written in them;
14. for "agile software development" and "extreme programming" styles of software development, HLLs have been the only game in town, even

though they can be seen to be a source of bottlenecks and associated project delays;

15. integration of separate software utilities—such as data-base and spreadsheet software—requires serious development effort over years/decades;
16. clumsy online interactive documentation for developers and users; only clumsy online access to rest of developing program code (viewing its text in a text editor), no database style access to it (where e.g. the system knows what the semantics of a particular variable is);
17. clumsy debugging tools for developers, difficult to modify as needed;
18. rapid prototyping difficult; prototype software not "malleable" into deliverable product;
19. when users run across bugs, it is difficult for them to communicate them to the developers, for many reasons;
20. legacy HLL to new HLL translators exist, but often the decision is a completely new re-programming effort;

21. *evolutionary* potential of standard HLLs practically exhausted, especially in a life-cycle cost-effectiveness sense;
22. although complex real-world problems have many possible dimensional-orientations/directions from which to analyze a given problem set and then design top-down structured programs, current HLL oriented programming systems do not allow more one of those to be implemented "cooperatively/-interactively"; this is somewhat like an engineer having to decide which of any number of possible orthogonal 2-D views is "the one", as opposed to today's 3-D CAD-CAM systems which allow just about any conceivable points and directions of view;
23. and many more.

ProgXML characteristics *will* eventually include:

1. scalability problems mitigated/postponed yet again; ProgXML has far more potential to be evolved to be more inherently scalable than standard HLLs, which have little or no such evolution potential;

2. potentially multi-dimensional (presentation and manipulation interfaces) semantics-*and*-pragmatics-based; the meanings of the program expressions are retained in their semantics-pragmatics markups; code re-adaptation/reuse can be made much more cost effective because re-adaptation/reuse is a *semantic* (and pragmatic) operation;
3. ProgXML interface guarantees “syntax” correctness as the program is entered and/or modified; examples include:
   - **a.** when entering strings or block of code, the text-box/form-field forces the string or block of code to be properly “delimited”;
   - **b.** variable identifiers can be chosen from a list, facilitating proper declaration and scoping, with variable typing checked/enforced at this time; comments, including ones that can be computer interpretable and applicable, are immediately available for each identifier, aiding semantic accuracy;
   - **c.** if variable identifiers are typed in directly, they are checked immediately for proper or

ambiguous declaration and scoping, with relevant feedback;

d. explicit intelligent semantics/pragmatics-based search/replace will not only guarantee "syntactic" correctness, but also greatly facilitate semantic-pragmatic correctness;

e. if the developer/programmer for some reason wants to leave a construct in an unfinished state, the system can "remember" this and automatically insert and/or substitute appropriate code if, for example, when the construct is included in a preliminary debugging/"shakedown" exercise;

4. ProgXML interface guarantees *some* (but not all) semantic(s) correctness as program is entered and/or modified (beyond e.g. typing of variables);

a. a 2-dimensional (or *n*-dimensional) decision table can be checked for rudimentary logical correctness;

b. programming (some) objects lends itself to multi-dimensional and/or graphical or visual representation (see Moody, Daniel L., “The ‘Physics’ of Notations”);

5. ProgXML semantic-pragmatics, presentation-markup languages, and systems are very easy to *extend*; extensions would not necessitate anything as complex and costly as extending an HLL and its compilers;
6. program semantics (and pragmatics)—at all levels—is held in ProgXML “Programming-semantics-pragmatics-extensible-markup” form, greatly facilitating access at those same informational-semantic-pragmatic levels; developer doesn’t have to read-interpret millions of lines of e.g. C++ code in text files to decipher semantics as s/he does now;
7. easily developed/extended interactive interfaces through ProgXML-driven dynamically one/two/-multi-dimensional forms/fields that are GUI in addition to current text-field-type screen-forms for entering data into XML databases; so, theoretically and practically, we must start thinking in terms of multi-dimensional syntax for the “ProgXML language(s)” (e.g. their input-presentation routines), as opposed to traditional

one-dimensional syntaxes associated with standard HLLs; examples:

a. 2-dimensional decision tables can be entered very simply using an interactive GUI; 3-dimensional decision tables are also relatively straightforward;
b. 2/3/?-dimensional graphs (for input-output) can be programmed or input 2-dimensionally ("piecewise" and "orientatably"), and output as currently (see Moody, Daniel L., "The 'Physics' of Notations");

8. easy to take advantage of a Dijkstra-style approach of designing input-presentation markup languages/interactive-GUI "languages" to solve *classes* of problems and then express particular solutions that are dynamic and "malleable" with respect to the evolution of problem perception (successfully to the extent that the "class-ification" and evolution-extrapolation analyses were done well);
9. XML-based interface can offer extensive/-extensible "hand-holding" for entry/viewing/-modifying/debugging of what are now standard HLL syntax and initial semantics-pragmatics

levels, even to the extent of wizards and/or tutorials that can be invoked dynamically, for e.g. objects in general, especially complex library objects, setting up "DLL"/"API" calls, etc, in addition to having database style access to existing code (e.g. libraries and programs already existing or under development) and its semantics-pragmatics (it was always "obvious" to many besides this author that the concept of a "database" developed extensively early on should be extended beyond data to programs, program code and programming, but… it never happened; now it can happen);

10. likewise, use of e.g. objects can be initially tested at entry time, in play, test and/or tutorial modes;

11. input of more complex objects (OOPS) can be made much easier with ProgXML hand-holding, e.g. with GUI;

12. semantics-based "source" allows ProgXML encodings that can be dynamically optimized for various qualities, such as adaptability to a given class of hardware/configurations or to a given class of presentation markup rendering "languages" (e.g. to help people who still need a

C# or JAVA style interface to the program); this base/mode retains the "values" and "meanings" of programs at all semantic levels that ProgXML has been extended to; this should allow reuse of the *semantics* at *all* levels, from highest to lowest, which should be much more efficient/ globally optimal than trying to reuse the syntactic description of fewer and lower levels of the semantics;

13. multi-level implementation concept "languages" (interactive "forms", as above) interrelating non-hierarchically to allow the program to be conceived, interacted with, and developed-evolved at the highest to the lowest levels "simultaneously", e.g. no/less need to discard high-level concepts in developing or generating "low-level implementation/code";

14. complex real-world problems have many possible dimensional-orientations/directions from which to analyze a given problem set and then design "top-down structured views" of a possible program; a ProgXML system would allow multiple such "presentation/interaction" views to be pursued in "cooperative/interactive" fashion; one could make some changes from one presented point of view, and other changes from another, and those changes could be checked by the ProgXML system in much the same way as modern CAD-CAM systems allow simultaneous changes from different engineering points of view and can "immediately" find inconsistencies deriving from such specification changes to different subsystems of e.g. a Boeing 777 entered by different engineers, as well as tracking the evolution and (hopefully) extinction of such inconsistencies and other bugs; this is a major evolutionary step forward in our concept of programming;

15. a ProgXML system is naturally synergistically compatible with just about any/all reasonable software development methodologies: agile software development, Extreme Programming (XP), Structured Systems Analysis and Design Method (SSADM), etc. The benefits of their strong points should be easy to improve significantly, and their weak points should be much easier to avoid/overcome using ProgXML.

16. variables etc. can simultaneously have different/multiple "levels" of naming: standard text names for various naming conventions (e.g. Hungarian-Simonyi), shorter mnemonic/-abbreviations for condensed display, nicknames, expanded descriptive names; developer can switch from one to another e.g. by hovering cursor, selecting option, etc.;
17. overall system "malleability", i.e. easy to modify extensively but safely, at all levels, over entire usage arena;
18. "integrateability" with other code from any platform at many levels of *semantics*, in addition to being more portable than traditional HLLs, regardless of the input-presentation markup rendering "languages" (e.g. C#) that were used by the developer to generate the ProgXML-based program code;
19. e.g. it should be feasible to integrate e.g. Office software with workflow and business process management software (even third party);
20. newer techniques like Unified Modeling Language and Model Driven Architecture are even easier to blend into a ProgXML

environment; a ProgXML environment can be designed so as to minimize the de-synchronization between the model and the actual program, currently a major problem;

21. the software development styles known as "agile software development" and "extreme programming" can benefit hugely from a ProgXML approach, which can both improve their strong points and mitigate their weak points;

22. relatedly, ProgXML offers many more possibilities for "malleable" semi-automatic modes, e.g. for dynamically combining automatic code generation and manual tweaking through some code-semantics-pragmatics variant of "hinting" or other cross-level directives semi-automatically/manually embedded in the ProgXML, either as part of initial ProgXML platform, or by developers in the program under development;

23. code-generation tools and 4GLs (fourth-generation languages) are very easy to design into a ProgXML platform and integrate with other approaches;

24. *many new* possibilities for more effective code optimization (for size, speed, etc.) at higher levels (e.g. algorithmic levels) and more globally than ever before; end that all too common combination of *both* code bloat and slooowww code;

25. integrated testing, debugging, diagnostics and performance testing, monitoring and optimization, standards or requirements adherence tools, and even project management tools can easily be built into a ProgXML platform, greatly reducing need to place any such directly in program code;

26. ProgXML-data/program-base-type tools can easily give much more of a picture of a ProgXML program or program suite and its functioning than any current tools can give of text based programs/suites;

27. Incremental/just-in-time compiling and linking/-building/executing should be easier than ever;

28. development infrastructure can be readily set up for centralized or distributed-cooperative development and management; (parts of) this same infrastructure can optionally be made part of the delivered product; since ProgXML databases can be made accessible and modifiable from all over the world, distributed interaction with a ProgXML system/database/-program should be straightforward;

29. a ProgXML system readily allows an important variant of scalability: simpler user interfaces for smaller-simpler development projects/-programs, and more complex user interfaces/tools *only as needed* for handling the special needs of larger and/or more complex programs, multiple distributed developers, integrated project management, etc.;

30. integration of separate software utilities—e.g. database and spreadsheet—will NOT require huge development effort over years;

31. rapid prototyping/developing can be a reality, with “requirements” that have been input into a ProgXML system “malleable” into prototype software, and prototype software “malleable” into deliverable product;

32. “specification languages/programs” (evolved into ProgXML versions) that can be executed to help realize verification and validation are a natural for ProgXML platforms; making such "malleable" so that the initial specifications can be incrementally and semi-automatically transformed into final/deliverable programs that meet performance requirements while maintaining specified functioning will be much more feasible with ProgXML platforms than currently;

33. the one-time (1960s and '70s) Holy Grail of inputting particular hardware architecture specifications into the compiler along with the program and have it spit out the executables for that processor and its peripheral chips (etc.; a variant of the Universal Turing Machine approach) starts to look more feasible with a ProgXML approach; this could similarly help interface to pre-existing "virtual machines";

34. linking/building instructions can allow multiple variants of routines—e.g. indirect-threaded-interpretive versus "hard-coded"—to be linked and/or run *alternatively* ***or collectively-simultaneously*** in situ with the inputs and outputs compared, resource usage and performance compared, etc; for example, there could be a fast version of a routine, a "guaranteed bug-free" version, a small-footprint version, etc;

35. when users run across bugs, the program they are using can easily be made more immediately helpful in both tracing and *communicating* the bug/suggestions to developers;

36. the problem of ProgXML-izing legacy HLL program source code is considerable, but will be much more feasible than currently re-writing e.g. COBOL programs in C++; but using only the desired parts of a favorite HLL for input/output presentation without having to inherit the Procrustean Bed of the entire HLL and its compiler is quite simple (e.g. one can use a legacy For-loop syntax or Object syntax from C++ to input such or see already entered program semantics output-presented in that syntax);

37. since ProgXML would (at least be extendable to) use e.g. C# or JAVA as a presentation form for the user to view and manipulate the program code (held in XML form), it would also (be extendable to) be able to output compilable C#

or JAVA source code, which is likely to be desirable early in the evolution of ProgXML;

38. ProgXML has great long-term evolutionary potential, especially in a life-cycle cost-effectiveness sense;

39. a usefully working ProgXML system should be easy to implement incrementally, with the first milestone being bootstrapping to where ProgXML is implemented in itself; thereafter, system evolution would be rapid due to ProgXML's inherent "malleability", "agility", "reusable patternability", "source code" manageability, and general ease-of-use;

40. our paradigms for computation are still largely based on "calculate-and-halt-type" programs, where e.g. "scope of variables" and "persistence of data-structures" are very simple, and very few of today's programs are of that type; exploring and experimenting with newer and more appropriate computation paradigms will be much more feasible with ProgXML;

41. the emphasis here on pragmatics in addition to semantics in ProgXML—as opposed to the predominantly syntactic approach of standard

HLLs—deserves emphasis; it is still too difficult to readily foresee the numerous ways being able to easily handle pragmatics issues more or less directly in addition to directly handling semantics issues will be found to be beneficial;

42. since a ProgXML system can always fall back on using given HLLs for input and presentation, the theoretical worst case of ProgXML programming system is no worse than for a system comprised of those same given HLLs;

43. and many *many* more.

Describing standard HLLs as high-cost, high-maintenance dinosaurs may seem cute, but it is all too true. Their truly useful days are numbered.

The above brief description of the possibilities of a genus-species of programming systems based on the concept of a "Programming-XML"—"ProgXML"—is just a hint of one of the inevitable waves of our software future.

With a right team, the initial research phase would ideally be 2 to 3 years, with the third year acting simultaneously as a "plan to throw one away; you will anyway" pseudo-development phase. The fourth year should see the development of a product that wouldn't be embarrassing to ship to beta users/general public. Un-ideally (think Windows version 1), a product could be produced in 2 to 3 years.

A right team should be feasible to assemble, provision and launch at (e.g.) Google. The research needs a great Arthur, ready to pull this sword from the stone. I would be a great Merlin (a retired software architect, middle 60s, unfortunately not in the best of health), especially having gestated this complex of ideas for years (even decades).

References

Moody, Daniel L., “The ‘Physics’ of Notations: Toward a Scientific Basis for Constructing Visual Notations in Software Engineering”, IEEE Transactions on Software Engineering, VOL. 35, NO. 6, Nov/Dec 2009