Thoughts on a Linguistic Infrastructure for Mission-Critical Software Development

Nicholas Weininger

May 26, 2001

1 Introduction

In recent years our lives and livelihoods have come to depend more and more on automated control systems. Historically, designs for such systems have been based upon redundant, well-understood hardware; software has been something of an afterthought. However, automated controllers in many domains, notably avionics, are now being called upon to perform increasingly complex tasks in software. The software developed to perform these tasks typically has two common characteristics:

1. it is developed largely using *"off-the-shelf" programming tools*, i.e. languages and programming environments designed for general-purpose software development;

2. however, unlike general-purpose software, its correctness is verified by a *formal certification process* which utilizes code review, coverage-based testing, and traceability of code to formally specified requirements.

The certification process ensures that life-critical software is not subject to the robustness problems experienced by, for example, Microsoft Windows. But as the complexity of software grows, this certification process becomes much more difficult, more costly, and less effective. In particular, manual review and testing become extremely time-consuming and error-prone. The state of the art in addressing certification complexity often involves replacing or supplementing manual review activities with automated verification tools of various kinds. The languages and environments used to develop the software, by contrast, are usually seen as more-or-less fixed constants.

In this paper, I contend that current approaches are incomplete, and that a real solution to certification complexity demands a radical rethinking of the linguistic infrastructure used to develop mission-critical software. Here I define "mission-critical software" fairly narrowly, to mean software used in situations where a malfunction will result in injury or death, since it is this sort of software which is usually subject to formal verification.

I argue in particular that the use of existing text-based programming languages impedes reviewers' understanding of software, and is an obstacle to the formal mathematical verification techniques required to catch subtle design errors. Because modern software programs have a fixed representation as textual source code, it is often difficult to organize a program in the ways best suited to certifying its correctness.

One alternative, a *representation-independent object-oriented visual language,* holds out the promise of increasing ease of verification dramatically, while remaining practical for large-scale software development. Such a language would be conducive to the building of small, reusable, easily-certified components; the intuitively understandable, visually expressible combination of those components; and the verification of a software system in terms of formal assumptions on component correctness. I will describe the basic structure such a language would have, and how its features would aid in developing certifiably correct software. No such language currently exists, but a project aimed at creating one is underway: the Eidola project developed by Paul Cantrell and sited at http://eidola.org. The rest of this paper is essentially an extended plug for applying the Eidola concept to the mission-critical domain.

Although Eidola would be particularly beneficial to mission-critical systems, the benefits it offers apply to all kinds of software engineering. This generality is essential to Eidola's success in the mission-critical domain, because mission-critical software developers tend to be leery of special-purpose solutions. Special-purpose languages and development tools tend to be constricted by their intended domain, hard to maintain, and subject to high costs of software purchase and training. Thus Eidola can only become usable for mission-critical development if it gains acceptance in "mainstream" software development.

The challenge in applying Eidola to mission-critical systems, then, is to simultaneously retain its general applicability and keep it compatible with the mission-critical domain. Ironically, Java, the modern programming language closest to Eidola's vision, is peculiarly unsuitable for some types of mission-critical development. Eidola, therefore, should have an object-oriented design at least as clean and sensible as Java's, without necessitating the introduction of those features which make Java unsuitable.

The rest of this paper is divided into five parts. First, I describe how the nature of existing programming languages causes problems for software verification. Second, I discuss in more detail what a representation-independent object-oriented visual language really means, and give some idea of how Eidola might implement the general principles of such a language. Third, I discuss how Eidola's capabilities could alleviate the problems of verification discussed in the first section. Fourth, I list necessary characteristics specific to a "mission-critical Eidola", that is, features that Eidola must have in order to work in the mission-critical domain. Finally, I discuss the implications of Eidola for the relationship between mission-critical software development and general-purpose software development. Readers who are already familiar with Eidola may wish to skip the second section; those who have experience developing and certifying mission-critical software may wish to skip the first.

2 Software verification and its discontents

> For a successful technology, reality must take precedence over public relations, for Nature cannot be fooled.
>
> –Richard Feynman

There exist several different sets of certification standards for different varieties of mission-critical software. My own (admittedly limited) experience is with the DO-178B standards for certifying software used in avionics, and so I will refer to those standards in the following discussion. However, the certification methods discussed, and their attendant problems, are generally applicable.

The DO-178B certification process essentially comprises the following steps:

1. *Formal requirements specification.* The tasks the software is to perform must be described completely and clearly in requirements documents.

2. *Code traceability to requirements.* Every line of source code must aid in implementing one or more of the specified requirements– i.e. "trace" to requirements– and every requirement must be implemented in code.

3. *Independent review of code correctness.* All source code files must be reviewed to ensure that code is written according to previously specified coding standards, and that the source code correctly implements the requirements it is supposed to implement. The reviewer for a particular segment of code must be someone who has not written that segment of code, so that every line of code is examined by at least two people.

4. *Independent testing for structural and requirements coverage.* Testers must develop a reproducible test suite to verify that the software actually performs the required tasks correctly. In the course of executing the test suite, every machine code instruction in the compiled object code must be executed, and every conditional branch must be both taken and not taken– i.e. "structural coverage" of all object code paths must be achieved. The testers must work independently of the software development team. Furthermore, the test suite itself must be reviewed independently for correctness.

Formal compliance with this process is a well-understood and relatively straightforward, if tedious, business. *Substantive* compliance is a different matter. It is one thing, for example, to show that each requirement in the specification has some block of code that traces to it; it is quite another to show that the relevant blocks of code actually satisfy the requirements they trace to. Reviewers and testers face a variety of difficulties in ensuring that a program purporting to satisfy a requirements specification actually satisfies both the letter and spirit of the specification.

2.1 Problems faced by reviewers

The principal substantive problem a reviewer faces is understanding how the source code implements a given requirement. In a simple system, there may be something close to a one-to-one correspondence between, say, class methods and requirements; but in a more complex system, the correspondence between requirements and code is often far from straightforward. Because a complex software system typically has a multilevel design, the code that satisfies a given requirement may be spread across multiple files and class definitions.

The structure of the parts of the system that satisfy a given requirement is not intuitively apparent from looking at any one code file, since a code file is by nature a low-level representation of a program design, and structural understanding requires a higher-level representation. The requirements document may attempt to ameliorate this by explaining the implementation structure using flowcharts or other graphical devices, but then the reviewer must verify that the textual source code really corresponds to the flowchart. The arrangement of this correspondence typically depends on the coding style of the development team, and it is likely to be easily comprehensible only to members of that team.

In practice, then, if you are a reviewer of a complex piece of software, it is extremely difficult for you to grasp the structure of requirements satisfaction in a program unless you have already worked on some part of that program. A "fully independent" outside reviewer may give a formally correct but substantially worthless review because of this lack of understanding. Thus, the best course is often to assign review of a given code file to someone who is on the program development team, but has worked on different code files. The network interface developer for a control program, for example, might review the program's fault handling routines.

This has two important negative consequences. First, it can compromise the intended effect of reviewer independence: a reviewer close to the development process is more likely to fall prey to hidden assumptions and comfortable shortcuts than is someone who has never seen the program before. Second, when developers must divide their time between formal certification reviews and development work, both development and certification slow down enormously.

Furthermore, most of the code in any given source file has nothing to do with most of the things a certification reviewer must check for. For example, common review tasks may include checking for array bounds violations and divisions by zero; in code that includes no array indexing or division operations, these checks are irrelevant. However, in order to verify this irrelevance one must actually read through the entire code file– a careless skim may miss a division or array index that you didn't think was there! The fact that so much review time is spent doing tedious checks for irrelevant things leads to an "eyes glazed over" problem, compromising reviewers' ability to think alertly about the things that really matter.

2.2 Problems faced by testers

It seems obvious that testers should be less affected by the textual representation of source code than reviewers. After all, independent testers do not need to look at the source code; indeed, they are usually prohibited from looking at the source code. Instead, a tester should treat the compiled program as a "black box" which must be tested against a requirements document, and neither the requirements document nor the object code should be affected by the source language's representation.

This is in fact true if the requirements are arrived at in advance before code development begins, and do not change during code development and testing. That is how things are supposed to happen. In the real world, code and requirements are more commonly developed in parallel, as the release of one version of a program breeds user demands for new features to be included in the next version. Indeed, code may be developed for "beta release" purposes before requirements are written for that code to satisfy. In such cases, some developer must determine the testing impact of new code: that is, he or she must determine what new requirements must be written and what existing requirements may be affected, as well as the changes in paths that may affect the achievement of structural coverage.

A developer working with testers thus needs an understanding of the impact on requirements and paths of a change in their code, just as a reviewer needs to understand the impact of a line of code on requirements satisfaction. Here, again, what is needed is a high-level view of the program structure and the "diff" of that structure produced by a code change; and again, the textual representation of source code makes it more difficult to comprehend that structure.

There is a larger problem, however, that is specific to testing. One may write a test for each requirement that verifies that that requirement is satisfied by the program– *under the conditions of the test suite*. However, that is very far from verifying that the requirement will be satisfied under *all possible* conditions that the program might face. For example, a network protocol might successfully deliver a packet within a time limit when a particular sequence of other packets have been delivered; but some other sequence of previous packet deliveries might cause the test packet to miss the deadline.

Testing for structural coverage does not solve this problem, because exploring every path in a program does not mean exploring every possible state of the program. Testers may find it next to impossible to produce a complete list of program states that need testing. In principle, most complex programs probably have an effectively infinite number of states, especially if *time* is a part of the program state. In practice, most of those states may be redundant for testing purposes; but since looking at a program's source code provides few clues to the structure of its state space, it is difficult to know for sure which states are redundant.

2.3 Solutions and their problems

2.3.1 Modularity

Among the "standard" solutions to certification complexity and cost, the most popular is a well-established technique of mainstream software engineering: building programs in a highly modular, object-oriented manner. This can work on two levels from a certification perspective. On a low level, modular construction of a program can break down the complicated review process into more manageable chunks: instead of verifying that requirement X is satisfied by a system all at once, a reviewer might initially verify that if classes A, B, and C all work as documented then the requirement will be satisfied, and then separately verify the correct operation of each class. On a higher level, a complex system can be separated into several mutually interacting, reusable components which are certified separately, with requirements on the correct operation of each component and on the inter-component interfaces. This reduces long-term certification cost by allowing components to be reused without recertification.

However, most existing programming languages usable for mission-critical systems are not really well-designed for modularity. Effective use of modularity to reduce verification costs requires strict adherence to encapsulation rules: for encapsulation enables easy verification that components do *not* depend on each other except in certain specified ways, and so allows the separation of review tasks for different components. But in some currently used languages, the object inheritance and typing syntax is not powerful enough, or not "clean" enough, to implement a complex design without breaking the encapsulation rules. C++ is a particularly glaring example of this failure. There exist several powerful programming languages which have good modularity properties, but they are generally either not suitable for mission-critical systems (e.g. Java), or else not sufficiently well-known and well-supported (e.g. Smalltalk).

Also, as stated in the previous sections, the textual representation of source code makes it difficult to understand the modular structure of a program by looking at code files. Thus it can be very tricky to show that the satisfaction of a given requirement follows from the correct operation of components. This task can be made easier by placing "implementation requirements" in the requirements document for a system, which specify the correct operation of class methods or other relatively small components, and then explaining in the requirements document how the "implementation requirements" work to satisfy the real system requirements. However, doing this increases development cost, since the requirements document often needs to be modified to correspond to code changes, even when the real requirements have not changed.

2.3.2 Annotating code with assertions

Another common verification technique, similar in spirit to modularity but working at a lower level, is the use of assertions to declare conditions that should be true at specified points in the operation of a program. Such assertions may be placed in comments in function bodies, function declarations, or class dec-

larations. They may take the form of invariants, which are supposed to be true at all times, or pre/postconditions which are to be true before or after a function invocation. Sometimes they are stated in a formal or semi-formal mathematical language. For example, the introductory algorithms class I took as an undergraduate included a verification of the correctness of a quicksort implementation, using a variety of formally stated preconditions and invariants.

The intent of such assertions is twofold: to make developers' thinking about the correctness of their design more rigorous, and to aid reviewers in understanding why an implementation is correct. If the proper tools are available, assertions may also be verifiable automatically, simplifying the human review process. Automated verification of assertions can reduce the amount of mundane review activity required and thus alleviate the "eyes glaze over" problem discussed in Section 2.1.

But automated assertion verification directly from source code is not yet practical, as discussed further in Section 2.3.3. Manual verification of assertions is challenging in part because it is difficult to know when assertions are sufficient to guarantee requirement correctness: too few assertions can give a false sense of security, too many can actually increase the amount of mundane review activity. More importantly, assertions often act like low-level "implementation requirements" and are thus prey to the same problems as other types of requirements: it is hard to assess the impact of a change in code on assertions, and the structure of code that satisfies an assertion can be hard to understand. Also, formalism in assertion statements is hampered by the distinction between the formal semantics of a programming language and its representation in source code.

2.3.3 Formal methods

There is considerable current research interest in the use of *formal methods*, i.e. mathematically based software tools for automated verification. The range of such tools available is extremely wide: they include automated theorem provers, logic checkers, and mathematical modeling languages of various types, some domain-specific, others not. I will focus on one particular type of formal methods tool: general-purpose *model checkers.* A model checker generally takes a program description and translates it into a set of state machines; it can then verify that the program satisfies certain conditions by exhaustively but "cleverly" exploring the state space. One example of such a tool is SPIN; a description of SPIN and links to some applications can be found at http://netlib.bell-labs.com/netlib/spin/whatispin.html.

Model checkers are especially promising in mission-critical applications, for numerous reasons. They can supplement structural coverage testing by covering portions of the state space that a traditional test suite might miss, and can check requirements that may be difficult to test for. They can be used not only for checking of high-level requirements, but also for low-level assertions (see Section 2.3.2). They can be used to model partial designs during the development process, providing the ability to check the soundness of part of a program's

design before implementing the whole program. Also, they can act as simulation tools as well as verification tools, and can serve as debugging aids. For example, when SPIN detects an assertion failure, it can graphically display the sequence of program states that led to the failure; this is an enormous potential advance over debugging an executable program, since a state sequence is generally very difficult to capture from an executable.

Model checkers face several large obstacles to their real-world application. The single biggest obstacle is the "state space explosion:" complex programs, as noted in Section 2.2, tend to have intractably huge state spaces, and reducing them to an explorable size is a very hard problem. Language representation cannot reduce the inherent difficulty of this problem: it will have to be solved by research into better reduction heuristics, and by more powerful computers that increase the size of the explorable state space.

Language representation, however, does affect the potential of model checking. Model checkers typically require that a program model be constructed using a pseudo-imperative modeling language; SPIN, for example, uses the C-like language Promela. In order to use a model checker for verification, then, you need to be able to translate back and forth between the modeling language and the source code language, and to verify that this translation is correct. This translation, and the interpretation of results given by the model checker, must be done by someone who has experience with the model checker as well as the program to be translated. For these reasons, the cost of using model checkers in development is often prohibitively high even if the state space explosion problem can be solved.

There has been some recent work on model checkers that work directly from source code. While promising, this work does not fully solve the problem of integrating model checking into the development process. For example, to use model checking as a debugging tool, the representation of a state sequence ought to be tied directly into the program development environment; state space illustrations ought to make reference to the actual program code. This is an as yet unexplored frontier. Furthermore, model checking directly from source code is hard when the source code language design is not mathematically "clean," as is the case with most current mission-critical development languages.

3 The Eidola concept and its realization

> Representations of the language thus exist for the benefit of the programmer rather than the compiler, and freed from the constraints of text files, we can tackle the question of how to notate a program *well.*
>
> – Paul Cantrell, from the Eidola webpage

The documents written by Paul Cantrell, available from the webpage http://eidola.org, include an exposition of the design motivation behind Eidola. I will give here only a brief and somewhat formalistic capsule description of Eidola's key characteristics; I urge interested or confused readers to go and read Paul's better and more detailed explanation. Keep in mind also that Eidola is still in the "pre-alpha" stage of development as of this writing, and so many important parts of it are still pipe dreams.

In the introduction to this paper, I defined Eidola as a representation-independent object-oriented visual language. Representation independence means that the fundamental form of a program is not defined by its representation to the programmer. Rather, an Eidola program "is" a set of abstract expressions conforming to a mathematically described formal semantics. An Eidola program is valid if and only if it conforms to the semantics, no matter how the program's component expressions are represented. Thus the formal semantics document for Eidola is not just an adjunct to a source code syntax; it is the complete specification of the language standard.

Object orientation implies that Eidola should have a clean, powerful typing structure providing the features we expect in an OO language: method encapsulation, inheritance, etc. Unlike certain other languages in which the OO design is grafted on to the algorithmic semantics as an afterthought, in Eidola the semantics of class definitions and inheritance are the first things to be laid out, and the features on which the most thought has been expended. The representation independence of Eidola means that it need not sacrifice a clean OO design in order to provide "syntactic sugar:" simplifying complicated general constructs, or providing useful shortcuts, are tasks that can be left to the representation.

The "visual" part of the definition means that some sort of visual notation scheme is needed to allow a programmer to view and edit Eidola programs. No such notation has yet been designed, although Paul has had some thoughts on the subject. The principal idea, of course, is that a notation should be superior to, or at least as good as, a text source code language. As discussed in Section 2, the major failing of textual languages, and the one that inspired Eidola's creation, is their failure to effectively represent the multilevel structure of complicated programs: a code file tells you only about the lowest level of structure.

This is a recognized problem in software engineering, and there are numerous existing tools designed to deal with it. Most of these tools are "wrappers" of various types around source code: class browsers, RAD tools, and the like. An Eidola notation may well resemble one of these tools; certainly it should in-

corporate the insights gained from their development. But representation independence means that an Eidola notation would be fundamentally more powerful than these tools: it would have full editing capability, unlike class browsers, and it would be free of translation/back-translation problems, unlike RAD tools. Furthermore, many visual development tools are domain-specific; an Eidola notation might also be domain-specific in its interface design, but the underlying programs it produced would be fully general and editable in other notations without change.

Of course, representation independence also opens the door to a wide variety of "exotic" notation ideas: 3-D structure walkthroughs, touch-based notations for the blind, etc. Nor is it necessary to assume that a program would be developed using a single notation. One notation could be tailored to the needs of initial design and prototyping, another to a "rapid-iteration" phase of quickly changing development, and still another to review and formal verification (on which more below). Eidola's representation independence ensures that multiple notations could work together on the same program without conflict; because the underlying abstract form of the program is the same for all representations, no one notation can compromise its generality.

Finally, it is worth noting that the Eidola semantics is, and will remain, an open standard; its reference implementation will be free software, and it will work with free tools whenever possible.

4 How Eidola could make verification easier

> The truth of the theorem is obvious from the following diagram...
>
> – Richard Lyons

I believe that Eidola could significantly improve the development and verification of mission-critical software, and in particular, that it could alleviate most of the problems enumerated in Section 2. First, because Eidola will be visually notated, it will make structural understanding of programs easier, and thus make code review easier and more effective. Second, because the fundamental form of Eidola programs is abstract and mathematical, it will make review automation and model checking easier. Third, the combination of abstract representation and visual notation will make for easy integration of the development and modeling processes, enabling developers to verify their designs more often and more fully. Each of these advantages deserves detailed examination.

4.1 Eidola's effects on code review

A good Eidola notation would, first and foremost, help reviewers by providing structural insight into how a program satisfies its requirements. The key advantage here is that a structural illustration of the program would be developed *as a part of code development*, not separately from it. Thus, unlike flowcharts or other adjunct illustrations, the notational representation of the program structure would require no additional effort above and beyond the code development, and it could never get out of sync with the program itself. Furthermore, when two versions of an Eidola program need to be compared, the Eidola notation could provide a visual "structure diff"; this would illustrate what really changed between the versions more effectively than any text-file diff could. Programmers have long recognized that good code should ideally be "self-documenting"; Eidola would bring that ideal closer to reality.

A notation designed specifically for formal verification could do even better. For example, such a notation might provide for hyperlinks from source code to specification documents. It might be able to automatically generate a report on traceability to documentation, or a list of the branch paths which need to be tested for structural coverage. Such helpful functions could be added on as "hints" annotating a program, without affecting the accessibility of the program code to other notations.

Because an Eidola program's structure would be easier to understand, it would be easier for a "fully independent" reviewer– one who is not familiar with the developers' coding style and has never seen the code before– to review a program effectively. This would rectify the problems with reviewer independence described in Section 2.1. Furthermore, while an Eidola notation could play the role that "syntactic sugar" constructs play in existing programming languages, it could also do away with the "syntactic cruft"– semicolons, for example– that is necessary for easy text processing but does not aid in program understanding. Eidola notations could use visual cues to bring out the important constructs in

a program, allowing a reviewer to concentrate on the more subtle conceptual review tasks without being impeded by notation.

Also, because the fundamental form of Eidola programs is abstract, automated or semi-automated analysis of semantic constructs would become much easier. Searching for, say, all division operators in a source code file is a tricky business at best, as anyone who has searched a C++ file for "/" knows. But a search for division operators in an Eidola program would be trivial, since Eidola programs are stored in terms of meaningful expressions. A good notation could present the results of such a search in a format that would make it (relatively) easy to evaluate, for example, whether any denominator might be zero.

4.2 Eidola's effects on formal modeling

The principal advantage of Eidola for the use of formal methods is that Eidola would alleviate the translation problem. Since Eidola programs are already in the abstract format that a model checker operates on, an Eidola program might be worked on directly by a model checker. Some additional constructs would probably be needed for modeling the temporal and hardware environment in which a software system exists, e.g. the temporal nondeterminism of a concurrent system. But adding those constructs would undoubtedly be easier than translating everything over to the modeling language.

An Eidola notation would also be well suited to integrating model checking with the development process. Assertions embedded in code could be checked either at runtime or by the model checker; when the model checker found an assertion failure, the code and data structures involved in the failure could be graphically called out as part of the program representation. If the programmer wanted to model a partial version of the program (perhaps because the full program state space was intractably large), he or she could do so, and then have the notation graphically illustrate which features of the full version were left out of the partial version. With text-based source code languages, this comparison is very difficult, especially when the partial version is heavily "cut down" and preserves only the high-level structure of the full program; in such cases, the source code file diffs are essentially useless, but an Eidola diff could call out the high-level differences effectively.

In particular, Eidola would aid the process of "program slicing." Very roughly, program slicing means extracting from a program the portions that are relevant to a particular condition or assertion. For example, if a programmer wished to assert that a given variable's value was always nonzero upon the invocation of a particular function, a program slicer might pull out just the constructs that set the value of that variable, and the scoping structure around them. Program slicing could be immensely helpful to the effective use of model checking, since it allows smaller, simpler "sliced" state spaces to be checked in place of the whole program's state space. But a program slicer needs to work on clean, easily analyzable abstract constructs, and getting these out of source code is a difficult task; since Eidola programs are already abstractly represented, slicing them ought to be easier.

5 Eidola for the mission-critical domain

> In theory, there is no difference between theory and practice; in practice, there is.
>
> – Dick Molnar

In order for Eidola to be viable for mission-critical applications, it must address not only the demands of verifiability, but also the task characteristics of these applications. The tasks performed by mission-critical software tend to have two peculiar attributes:

1. they are *real-time*; that is, they must be guaranteed to complete execution within a specified time interval after initial invocation, often a periodic invocation with a fixed period (e.g. once every 20 milliseconds).

2. they are *embedded*, i.e. dedicated to the control of special-purpose hardware, and so need to be able to communicate directly with that hardware– typically through references to fixed memory addresses or ports. This also often means that specialized subroutines need to be written in assembly language.

Thus, a mission-critical executable typically must not contain any routines– including runtime libraries or other support routines– that are not verifiably deterministic in their operation; and a language for mission-critical development must allow low-level interaction with hardware. These requirements are what make Java, for example, unsuitable for mission-critical development: Java contains nondeterministic garbage-collection routines as a standard, unremovable part of the language, and its design deliberately makes it difficult to reference memory directly.

However, features such as garbage collection and rich standard libraries are extremely desirable for most mainstream software development, and they will likely be included in Eidola. As stated in the Introduction, if Eidola is to be usable for mission-critical development, it must gain acceptance in the wider non-mission-critical world. Therefore, there will probably have to be a special "mission-critical" version of Eidola which is compatible with, but not identical to, the mainstream version. This special version, hereafter referred to as MC Eidola, should contain only those restrictions and/or additional features strictly necessary for the mission-critical domain; an MC Eidola program should be editable by any compliant Eidola notation, but may have a separate compiler and/or runtime environment.

I will first discuss the ways in which MC Eidola will need to depart from the mainstream version of Eidola, and then discuss generally applicable features that are especially pertinent to the mission-critical domain. The following should be taken as a set of recommendations for future design work; remember that Eidola is still in the early stages, and much nontrivial development needs to be done.

5.1 Special features of MC Eidola

The overriding necessity behind MC Eidola is the need to create a "pure" executable, i.e. one which strictly limits the scope of runtime support code that is not actually part of the Eidola program. Any runtime support code that does have to be included must be assured not to interfere with the needs of a mission-critical application. For one thing, this means that an MC Eidola program must be compilable to real machine code, not to a virtual machine.

It also means that MC Eidola will probably have to disallow the use of certain language features of general Eidola; garbage collection is the obvious example, but others have been suggested (e.g. runtime type information, dynamic class loading, reflection). The nondeterminism of garbage collection is really just one instance of the memory allocation problem; in fact, all memory allocation routines are likely to need modification in mission-critical applications. Different mission-critical applications tend to have different requirements for memory allocation schemes, depending on the environment in which they run. Some may even require special things to be done with stack allocation of local variables. Thus an MC Eidola development environment will probably have to provide hooks that allow developers to write their own application-specific versions of runtime memory allocation routines. It may also prove necessary to add a "delete" operator to the language which does nothing in the general version of Eidola but can be implemented nontrivially in MC Eidola.

There is at least one more language feature requiring runtime support code: method invocation. While this clearly cannot be removed from MC Eidola, the runtime support code necessary for it is simple enough, and limited enough in scope, that it will not present a big problem for verifiability. Furthermore, method invocation support code is already unavoidably present in almost all existing programming languages, so it is already a known problem for mission-critical developers.

The other major category of standard runtime code will involve standard class definitions for I/O, commonly used types, data structures and algorithms, etc. MC Eidola should simply disallow most of these; mission-critical developers can write their own verified versions if necessary. Again, since existing programming languages already contain such standard libraries (e.g. the C++ input/output libs), working without them is a known problem in mission-critical development. And again, in those cases where the runtime code cannot be omitted (e.g. definitions for very basic data types like integers and characters), it is simple enough not to present a problem.

5.2 General Eidola features helpful to the mission-critical domain

Eidola's algorithmic semantics should make it possible to write a function body in another language, perhaps as part of a library, and then call that function from Eidola code. This is a good thing for mainstream software development, but is especially important for mission-critical development, since many routines

in a mission-critical program may have to be written in assembly language. Note that such interfaces are very hard to do well, and doing them in Eidola is very much an unsolved problem. Dealing with calling conventions and side effects when calling functions written in other languages, for example, is very hard to do in a portable manner. From a mission-critical perspective, it is fortunate that such interfaces need to be in general Eidola, since the design effort required to include them might not be justified for mission-critical needs alone.

Finally, for Eidola to be maximally useful in developing verifiable software, an Eidola development environment should have a rich expression language for assertions that refer to program objects. Such assertions should ideally include quantifiers, e.g. "all Foobar objects in the system have a properly initialized Baz member"; quantified assertions would have to be checked either by a special runtime library or by a model checker. It is not clear at this point, however, whether such assertions would actually need to be part of the Eidola semantics proper. Assertions might instead exist as "hints" which would be attached to program elements but not affected by program semantics; a supplemental standard for assertion semantics could then function as an optional addition to the Eidola program semantics.

6 Conclusion

> The Great Shame of computer science is that we don't seem to be able to write software much better as computers get much faster... The distance between the ideal computers we imagine in our thought experiments and the real computers we know how to unleash on the world could not be more bitter.
>
> – Jaron Lanier

While I have dwelled in this paper upon the problems of mission-critical software development, in a sense the really remarkable thing is how well it works. Laments over the persistence of bugginess in software are legion; Jaron Lanier's, quoted above, is just one of the most recent. Yet, for example, the software which goes into our airplanes has a very good quality record; as of this writing, I can still confidently state that no plane has ever crashed due to a software bug. This is no excuse for complacency, but it serves to illustrate the unusually good record of the mission-critical domain, in an industry plagued by quality problems.

Enormous numbers of papers have been written debating the reasons for these problems, and proposing various schemes for improving software quality. Many of these schemes involve rigid, formalistic development processes that perhaps seek to emulate mission-critical certification processes. As I see it, such schemes miss the point. We really already know how to develop high-quality software, and it is common-sense practices, not formalized processes, that do it. The necessary practices are:

- formal analysis of design and implementation
- vigilant, independent code review
- extensive testing

The problem is that we don't know how to develop high-quality software *quickly and cheaply enough*. Analysis, review, and testing, when done right, require highly skilled developers and large investments of developer time. Since developer time is extremely expensive and the software market demands fast product lifecycles, most software developers are not able to employ common-sense quality practices to the extent they should. High-quality software is thus produced only when the criticality of the application justifies the high cost of these practices.

This is where the impact of Eidola could be most deeply felt. As I have attempted to demonstrate in this paper, Eidola could potentially achieve large increases in the ease, and therefore reductions in the cost, of review and analysis. A good Eidola system should make it easier to perform automated or semi-automated review and verification tasks informally, even when a formal certification process is not required.

If the good practices used in mission-critical development become easier and cheaper, they are likely to gain wider acceptance outside the mission-critical

world. At the same time, Eidola can bring to mission-critical development the design advantages of mainstream OO languages like Java. Thus Eidola could help to "close the gap" between mission-critical and other software, improving the quality of both.

Another frequently heard complaint in computer science is that few far-reaching software innovations have found acceptance in the real world: we still use graphical interfaces, for example, whose essential features were designed at Xerox PARC in the 1970s. I hope Eidola will serve to answer that complaint. It will not solve the many truly difficult problems of software engineering; it will not make it any easier to come up with better algorithms or better data structures. But it may free software developers to focus on the important problems, rather than spending their time on the mundane problems created by source code.