Submitted to SPLASH Onward! 2011

Designing ParaSail

A new language for race-free parallel programming

S. Tucker Taft

SofCheck, Inc.

Tucker.Taft@SofCheck.com

Abstract

ParaSail is a new parallel programming language whose design process over the past two years was documented by, and intertwined with, a web log. This approach to language design differs significantly from the more conventional process involving a language design team, or a language design committee. However, any language design process benefits from a strong, explicit, consistent set of design principles which guide the process and provide criteria to select among the multitude of design choices. This paper identifies some of the principles which drove the design of ParaSail, compares them with principles either explicit or implicit in designs of other languages, and attempts to evaluate how well such principles served the corresponding language designs.

Categories and Subject Descriptors CR-number [*subcategory*]: third-level

Keywords Programming Language Design, Parallel Programming, Design Principles, Inherently Parallel, Compile-Time Checking, Language Design Process, Weblog

1. Introduction

Designing a new programming language is always a daunting task. Designing a language that is intended to have broad applicability, advance the state of the art, and be easy and safe to use, makes the job even more challenging. To design ParaSail, a new language for race-free parallel programming, the author chose to partake this effort out in the open, in a publicly accessible weblog, but otherwise as a solo effort[1].

Prior work by the author included over 30 years involvement with the language design and standardization process, including 5 years as technical lead of a full-time design team developing the revision to the Ada language now known as Ada 95[5]. Other activities have involved more typical volunteer language design committee approaches, where there is no real *command* structure in the committee, requiring a somewhat different process.

The *committee* approach to design, without strong leadership, faces serious challenges in avoiding unpleasant compromises and feature inconsistencies. Perhaps the most famous quotation along these lines is that a camel looks like a horse designed by committee. Certainly some programming languages have acquired a few camel humps, if not in the original design, then as the language evolved.

2. Blog-based Language Design Process

The solo, *blog*-based process of language design opens up new opportunities, while creating its own set of challenges. Clearly the command structure is very simple, but the ability to validate ideas against other members of a team are clearly limited. The authors lengthy experience with language design is presumably an advantage, but perhaps the biggest advantage is having a very strong set of principles for what would make this new language most valuable to the software development community. These principles include the following:

- The language should be easy to read, and look familiar to a broad swath of existing programmers, from the ranks of programmers in the Algol/Pascal/Ada/Eiffel family, to the programmers in the C/C++/Java/C# family, to the programmers in the ML/Haskell and Lisp/Scheme communities. Readability is to be emphasized over terseness, and where symbols are used, they should be familiar from existing languages, mathematics, or logic. Although extended character sets are more available these days, most keyboards are still largely limited to the ASCII, or at best, the Latin-1, character set, so the language should not depend on the use of characters that are a chore to type.

  Programs are often scanned backward, so ending indicators should be as informative as starting indicators for composite constructs. For example, “end loop” or “end class Stack” rather than simply “end” or “}”.

- The language should have one primary way to do something rather than two or three nearly equivalent ones.

[Copyright notice will appear here once 'preprint' option is removed.]

Nonessential features should be eliminated, especially those that are error prone or complicate the testing or proof process. User-defined types and language-defined types should use the same syntax and have the same capabilities.

- All code should be parameterized to some extent, since all code being written in this day and age should be parameterized over the precision of the numeric types, the character code of the strings involved, or the element types of the data structures being defined. In other words, all modules should be generic templates or equivalent. But the semantics should be defined so that the parameterized modules can be fully compiled prior to being instantiated.
- Parallelism should be built into the language to the extent that it is more natural to write parallel code than to write explicitly sequential code, and that the resulting programs can easily take advantage of as many cores as are available on the host computer.
- The language should be inherently safe, in that the compiler should detect all potential race conditions, as well as all potential runtime errors such as the use of uninitialized data, out of bounds indices, overflowing numeric calculations, etc. Given the advances in static analysis, there is no reason that the compiler cannot eliminate all possible sources of run-time errors.

Perhaps much of this can be summed up in the words of the language designer Jean Ichbiah, namely, that programming is fundamentally a human activity [2]. Programming is about human programmers clearly and correctly communicating with at least two audiences: 1) other human programmers, both current and future, and 2) a very literally-minded machine-based compiler or interpreter. What is needed is *human engineering*, which is the process of adapting a technology to be most useful to humans, by minimizing opportunities for errors, taking advantage of commonly understood principles, using terminology and symbols consistently and in ways that are familiar, and eliminating unnecessary complexity.

2.1 Some lower-level principles

Here are some additional somewhat lower level principles followed during the ParaSail design:

- Conventional wisdom should not be accepted without challenge. In particular, though passing functions and types as parameters is clearly useful, it is arguable whether full upward closures and types as first-class objects, are useful enough to justify the significant testing and proof burdens associated with such constructs. The more disciplined packaging of type and function provided by object-oriented programming can match essentially all of the capability provided by upward closures and types as first-class objects, while providing, through the Liskov Substitutability Principle[3] and other similar principles, a more tractable testing and proof problem.
- Avoid constructs that require fine-grained asynchronous garbage collection if possible. Garbage collectors are notoriously hard to test and prove formally, and are made even more complex when real-time and multi-processor requirements are added. Region-based storage management, as in the *Cyclone* language[4], suggests a possible alternative approach.
- Mutual exclusion and waiting for a condition to be true should be automatic as part of calling an operation for which it is relevant. This is as opposed to explicit lock/unlock, or explicit wait/signal. This simplifies programming and eliminates numerous sources for errors in parallel programs with inter-thread synchronization. The result is also easier to understand and prove correct.

The design of ParaSail is now largely complete (after an approximately 18 month process), and a prototype implementation is now usable. This paper will conclude with an analysis of how well the design process worked, as related to the above design principles, and as related to the blog-based approach.

3. Programming Language History

It is interesting to catalog some of the mainstream, or at least highly influential, third generation programming languages designed over the past 45 years or so, and who were their lead designers:

```
1956 Fortran -- John Backus
1957 LISP -- John McCarthy
1958 COBOL -- Grace Hopper et al
1960 Algol 60 -- Bauer, Backus, Naur, et al
1964 BASIC -- J.G. Kemeny and T.E. Kurtz
1964 PL/I -- IBM & SHARE committee
1964 APL -- Ken Iverson
1966 Algol W -- Niklaus Wirth, C.A.R. Hoare
1967 Simula 67 -- O.J. Dahl, Kristen Nygaard, et al
1968 Algol 68 -- C.A.R. Hoare, Edsger Dijkstra, et al
1970 Pascal -- Niklaus Wirth
1970 Smalltalk -- Alan Kay
1972 Prolog -- Alin Colmerauer
1972 C -- Dennis Ritchie
1973 ML -- Robin Milner
1974 CLU -- Barbara Liskov
1975 Concurrent Pascal -- Per Brinch Hansen
1975 Scheme -- Guy Steele and Gerald Sussman
1975 Modula -- Niklaus Wirth
1980 Ada 83 -- Jean Ichbiah (at INRIA)
1983 Turbo Pascal -- Anders Hejlsberg
1983 Occam -- David May (at INMOS)
1983 C++ -- Bjarne Stroustrup
1986 Eiffel -- Bertrand Meyer
1986 Oberon -- Niklaus Wirth
```

```
1986 Modula-3 -- Luca Cardelli et al (at Dec SRC)
1986 Erlang -- Joe Armstrong (at Ericsson)
1990 Haskell -- S.P.Jones, P. Hudak, et al
1995 Java -- James Gosling
1995 Ada 95 -- Tucker Taft et al (at Intermetrics)
1995 Delphi -- Anders Hejlsberg
1996 OCaml -- Xavier Leroy
2001 C# -- Anders Hejlsberg
2003 Scala -- Martin Odersky
2005 Fortress -- Guy Steele et al
```

Although there have been exceptions, such as Algol 60 and 68, PL/I, Haskell, and Modula 3 (which were all designed from the beginning by a tight-knit committee), by and large these languages grew from the vision of one, or at most two, chief architects, some of whom appear in the above list several times with different languages, in different decades. It is notable that the bulk of these languages originated (at least in some form) more than 25 years ago, and all but the last two more than ten years ago. Of course part of the reason for the age of these languages is that it generally takes several years for a language to reach "mainstream" status, and once a language reaches such status, it is often an even slower process for it to lose its status, and open up room for some new upstart language. It is also interesting to see Fortran and Fortress as bookends on this list of languages, both languages largely focused on numerical calculation, while the languages in between have wandered far afield from a focus on numerics.

So counting distinct designers above, we would guess that probably 95% of all 3rd-generation programming over the past 45 years has been done in languages designed by a total of about 30 different people, and probably just ten people, Backus, Hopper, Kemeny, Kurtz, Wirth, Ritchie, Steele, Stroustrup, Gosling, and Hejlsberg, could be credited with the lions share of that.

4. Language Design Philosophies

Formal evaluations of the comparative strengths of languages generally try to be objective, weighing expressiveness, readability, maintainability, efficiency, portability, etc. However, given the very personal nature of most language designs, it is not surprising that languages tend to develop "followings" more than true rational justifications for their use. Most languages are Turing complete within the capacity of the execution environment, so expressiveness depends less on the language per se than on the fluency of the programmer with the language.

So what ultimately makes a language design popular, influential, or well regarded? Our view is that it is all about the integrity and consistency of the language design philosophy held by the designer or design team. Some language designers make it easy to identify their design philosophy, because they address this issue explicitly, and make an effort to explain the choices they made. Others make it harder, but often one can find a casual phrase in a language reference manual that might capture their attitude.

Bjarne Stroustrup states in the second edition of *The C++ Programming Language*[6]:

> The ["C with classes"] language was originally invented because the author wanted to write some event-driven simulations for which Simula67 would have been ideal, except for efficiency considerations. ... C++ was designed primarily so that the author and his friends would not have to program in assembler, C, or various modern high-level languages. Its main purpose is to make writing good programs easier and more pleasant for the individual programmer. (pp. 4-5)

Concerns for readability, maintainability, correctness, robustness, efficiency, etc., are all bundled here into the definition of what makes a "good program."

Bertrand Meyer gave hints of his design philosophy for Eiffel in *Object-Oriented Software Construction*[7] and *Introduction to the Theory of Programming Languages*[8]. In the preface to [7], Meyer states: "I designed [Eiffel] because no existing language was up to my expectations." He goes on to say "Some of the chapters ... include a 'discussion' section explaining the design issues encountered during the design of Eiffel, and how they were resolved. ... I often wished, when reading descriptions of well-known programming languages, that the designers had told me not only what solutions they chose, but why the chose them."

Both Stroustrup and Meyer later wrote books specifically about the design of their languages, and in there we can find explicit explanations, or in some cases after-the-fact rationalizations, of choices made during the design. In *The Design and Evolution of C++*[9] Stroustrup goes into great depth on particular C++ language features. He begins chapter 4 of this book with: "To be genuinely useful and pleasant to work with, a programming language must be designed according to an overall view that guides the design of its individual language features." But he then goes on to say: "I call them *rules* because I find the term *principles* pretentious in a field as poor in genuine scientific principles as programming language design ... if a rule and practical experience are in conflict, the rule gives way." In general Stroustrup seems to believe in the importance of having a set of rules, but also seems to have relatively little faith in the ability to develop a set of rules that can actually be applied consistently across the language. Here are some of his rules:

- C++'s evolution must be driven by real problems.
- Don't get involved in a sterile quest for perfection.
- C++ must be useful *now*.
- Don't try to force people.

Meyer in the preface to his book *Eiffel: The Language*[10], emphasizes the importance of leaving things out of a design:

> Also notable is the set of ideas that have *not* been retained. To design is to renounce.... In Eiffel, ... much attention was devoted to keeping the language small and trying to make it elegant...

The Modula-3 design team at DEC's SRC documented their underlying design philosophy. One of their key goals was a language specification of no more than 50 pages[11]. This doesn't say anything about maintainability, readability, etc. of the programs, but it does address the value of a small, internally consistent set of straightforward language features. Greg Nelson, a member of the team, also wrote a number of *How Modula-3 Got Its Spots* essays, which provide insight into the decision process used to choose particular rules for particular features[11][12].

In 1992, Barbara Liskov wrote an excellent description of the design process and principles behind the CLU language[13], which she and her graduate students at MIT designed in 1973. The identified principles were: 1. Keep focussed, 2. Minimality, 3. Simplicity, 4. Expressive power, 5. Uniformity, 6. Safety, 7. Performance. That is a pretty good list for almost any language.

The original Ada 83 reference manual[2] includes an introduction which identifies three overriding concerns in the design: "program reliability and maintenance, programming as a human activity, and efficiency." Because programming is a human activity, *human engineering* is important to minimize error-prone constructs. The fundamental challenge is that programmers make mistakes. From an engineering point of view, the challenge is similar to that of a noisy communication channel, and the desire to identify any errors in the transmission to avoid getting the wrong message. One chief technique is to increase the *hamming distance* between legal messages. That means that when some amount of noise is added to a message, it is likely to be recognized as an illegal message, rather than being interpreted as the wrong legal message. A related technique is increasing the redundancy in the message, where any noise is likely to appear as inconsistencies between the redundant parts of the message.

The Ada 95 language design team created an annotated version of the Ada Reference Manual[14], and within certain sections provided one or more paragraphs labeled *Language Design Principles*, to document the principles underlying the design of the features described in that section. For example, in the section on "context clauses" (which identify which other modules are to be imported by a given module and whether the components of the module are to be made directly visible), the following Language Design Principle paragraph is provided:

> The reader should be able to understand a context_clause without looking ahead. Similarly, when compiling a context_clause, the compiler should not have to look ahead at subsequent context_items, nor at the compilation unit to which the context_clause is attached.

By explicitly identifying these principles to users of the language, the rules of the language can become more intuitive to the programmer, and future evolution of the language is more likely to remain consistent with the original design philosophy.

An article by the author on the Ada 2005 language design process[15] emphasizes the importance of a shared design philosophy, with overarching concerns with "safety and efficiency, with safety given more weight – though never absolute precedence - when there was a conflict. ... [B]y appropriate human engineering, you can produce a language that is in the end more productive." This article concludes that the presence of a strong, shared design philosophy within the committee responsible for the Ada 2005 revision helped the committee avoid the know pitfalls of committee design.

5. Enumeration Types and the Blog-Based Design Process

One of the features of a blog-based language design process is that it can be almost intentionally non-systematic, where parts of the design that are clear can be described, while the parts that are fuzzy can be ignored completely, or can merely be identified as fuzzy and then left unresolved until some inspiration arrives. A committee-based or design-team-based approach can rarely afford to just ignore some important part of the language. In general a subcommittee will be assigned to "solve the problem."

As an example, deciding whether and how to provide enumeration types has been a challenge for several object-oriented languages. Wirth in his design for Pascal included enumeration types, and most language designers since then have included them. Interestingly, however, they were left out of Wirth's newer langauge Oberon in part because Wirth did not consider them "extensible" enough, and the Java designers also omitted them for apparently the same reason. Ultimately they were added back to later versions of Java (though not to more recent versions of Oberon).

Below is Wirth's explanation for why he omitted enumeration types. Based on experience using Java before enumeration types were added back, this seems a very short-sighted decision, because there is a fundamental and important distinction between an enumeration type and an integer type, which is lost in Oberon and initial Java. In any case, here is Wirth's explanation[16]:

> Enumeration types appear to be a simple enough feature to be uncontroversial. However, they defy extensibility over module boundaries. Either a facility to extend given enumeration types has to be introduced, or they have to be dropped. A reason in favour of the latter, radical solution was the observation that in a growing number of programs the indiscriminate use of enumerations (and subranges) had led to a type explosion that contributed not to program clarity but rather to verbosity. In connection with import and ex-

port, enumerations give rise to the exceptional rule that the import of a type identifier also causes the (automatic) import of all associated constant identifiers. This exceptional rule defies conceptual simplicity and causes unpleasant problems for the implementor.

Bertrand Meyer also chose to omit enumeration types from Eiffel. Here is his explanation[7]:

> Introducing Pascal-ilke enumeration types would be a conceptual disaster in Eiffel: they would conflict with the type system of the language, which is otherwise simple (the four simple types on the one hand, and the class types on the other). It does not seem feasible to combine this notion elegantly with inheritance.

In the design of ParaSail, though we knew that we wanted to have enumeration types, we were struggling to find a way to integrate a mechanism for defining enumeration types with the general model that had been adopted for all other type definitions, namely the instantiation of a module with parameters. For example, to define an integral "Age" type in ParaSail, one could write:

```
type Age is Integer<0..200>;
```

To define a set of Ages, one could write:

```
type Age_Set is Set<Age>;
```

To define an array type of Ages indexed by Employee_ID, one could write:

```
type Employee_Ages is Array
   <Age, Indexed_By => Employee_ID>;
```

In each case, we define a type by instantiating a module with particular parameters. The challenge with an enumeration type is to identify the particular enumeration literals to be associated with the type, while remaining with this general principle that one defines a type by instantiating a module.

Although we had high hopes for solving this problem, the blog-based design approach gave us the license to simply not worry about the problem until a solution presented itself. Had we felt forced to systematically "solve the problem," we almost certainly would have resorted to some kind of special syntax unique to the definition of enumeration types to accommodate identifying the enumeration literals, giving up on our uniform syntax for type definition.

While we temporarily ignored the problem with enumeration types and the associated literals, we proceeded to work out the details of integer, floating point, character, and string literals, and how they could be associated with a particular type. Our model was to associate a predefined "universal" type with each kind of literal, and introduce a special set of "from_univ" conversion operators, one from each universal type, which if defined for a particular user-defined type, allowed that user-defined type to make use of that kind of literal. The precondition on the "from_univ" operator determines what range of literals is acceptable, and the body of the "from_univ" operator performs the actual conversion of the literal to the user-defined type.

For example, to allow a particular type "My_Type" to use integer literals in the interval -100 .. +100, one would define a "from_univ" operator as:

```
operator "from_univ"
    (Literal : Univ_Integer
      {Literal in -100 .. +100})
    -> My_Type;
```

The above operator has one input parameter Literal of type Univ_Integer restricted by a precondition to the interval -100 .. +100, and with an output of My_Type.

This model seemed to accomplish a goal of giving user-defined types and language-defined types equal capability as far as literals. Having described it in the blog and played with it in various contexts, we moved on to other issues feeling that the notion of user-defined literals was well in hand.

Approximately six months later, a satisfying solution for the enumeration type with its enumeration literals emerged, somewhat out of the blue. With a committee or design team, this kind of delay for such a fundamental feature would almost certainly be unacceptable. Of course, it is possible that with more people focused on a particular problem, a similarly satisfying solution would emerge, but it seems a legitimate question whether one can simply "order up" satisfying solutions for any given design challenge.

In any case, the solution for enumeration types that emerged was to treat enumeration literals much as we were treating numeric, character, and string literals. All of the integer literals were of the single Univ_Integer type, and it was the availability of a "from_univ" conversion operator that determined whether a given user-defined type could be used with integer literals. The idea was to create a recognizable syntax for enumeration literals (distinct from that for normal program identifiers), and associated them all with a single Univ_Enumeration type, in the same way that integer literals were all of the Univ_Integer type. If a user-defined type was to be treated as an enumeration type, then it would require a "from_univ" operator providing a conversion from Univ_Enumeration, with a precondition indicating which particular values of the Univ_Enumeration type were acceptable as input to the conversion. For example, an enumeration type Color with three enumeration literals would have the following "from_univ" operator:

```
operator "from_univ"
    (Literal : Univ_Enumeration
       {Literal in [#red | #green | #blue]})
  -> Color;
```

The above can be read as: the operator "from_univ" takes an input parameter of type Univ_Enumeration and returns an output of type Color, with the precondition that the input parameter be in the set [#red | #green | #blue]. As illustrated, the syntax for a literal of type Univ_Enumeration is a '#' followed by an identifier. Clearly Univ_Enumeration has an effectively infinite number of distinct literal values, much as Univ_Integer has an infinite number of distinct literal integer values.

So how does this help with our desire to make the definition of an enumeration type use the syntax of a module instantiation? The answer is that we can define a module, perhaps called Enum, which takes a vector of Univ_Enumeration values, and internally defines a "from_univ" operator based on that vector, with the set of literals in the vector providing the precondition, and the order of the literals in the vector determining the underlying representation as, for example, an integer in the interval 0..num_literals-1. For example, if we presume the existence of such an Enum module (and there might be other modules for defining enumeration types with alternative representation approaches), then we can define an enumeration type Color via:

```
type Color is Enum<[#red, #green, #blue]>;
```

This approach depends explicitly on being able to write the relevant enumeration literals *before* defining the enumeration type. This is now possible because all enumeration literals are of a predefined Univ_Enumeration type, and can be used freely without ever declaring a user-defined enumeration type.

This solution is particularly satisfying, because it unifies enumeration type definition syntax with that of all other types, it gives complete freedom in how to represent enumeration types internally, and it addresses one of Wirth's complaints from the above quotation by allowing an enumeration type to be imported without having to *carry along* a large collection of other identifiers. The enumeration literals already exist everywhere, rather than being carried along with particular enumeration types.

6. Evaluating the Language Design Process

How does one evaluate the success of the design of a programming language? One measure is to observe the amount of literature devoted to helping programmers avoid problems with the language. With C being one of the most widely used languages applicable to many domains, it is not a great surprise that a long list of human engineering issues has been produced. Entire books have been devoted to the "pitfalls" of C[17]. Examples such as the "=" vs. "==" confusion, the missing "break" bug, operator precedence surprises, the missing "" challenge, the forgotten function "return" statement. Are such problems inevitable? C was certainly a big step up from assembly language, the language in which most system software was written prior to the wide availability of C.

Other languages have developed their own list of pitfalls. For C++, in addition to the list inherited from C, there is the notorious virtual vs. non-virtual destructor issue, the dangling "const" syntax for qualifying the "this" parameter, the type slicing which can happen when passing a class-type object by value, the meaning of "&" vs. "*" in declarator syntax as opposed to call syntax.

As an example of the kinds of issues found in the Pascal language, there is the ambiguity in the run-time semantics of "and" and "or," the confusion of semicolon-as-separator vs. semicolon-as-terminator, and whether to repeat or not the parameter declarations when a function has a forward declaration.

Are these kinds of problems avoidable during the design process? Does it help to have a strong, explicit design philosophy that includes human engineering? C and C++ were languages designed according to a relatively informal and pragmatic set of rules. By contrast, CLU and Modula-3 were designed with an explicit set of strong design principles. Clearly using these languages as data points, the long term popularity of a programming language is not determined by the strength of its design principles. But popularity of technology is notoriously hard to predict based on inherent characteristics. CLU and Modula-3 are widely regarded as having clean, elegant, internally consistent designs. Let us hope that does not always preclude commercial popularity.

7. Conclusion

Using a weblog to document and drive the design process for the ParaSail programming language has been, we feel, a successful experiment. What lessons can we draw from this experience? Having an explicit, strong set of design principles has been very important, even for this solo blog-based effort.

But in contrast to a more systematic design-team-based or committee-based process, the blog-based process has allowed us to jump from one aspect of the language to another, allowing the author to go through more of a *discovery* process than an *invention* process. That is, rather than systematically tackling particular design problems and attempting to force a solution, problems were allowed to percolate in the background while effort focused on parts of the language where solutions were more immediately apparent. At some point, as part of experimenting or ruminating, a solution for one of the background problems suddenly emerged, as though it was always there but just was not yet visible. This has allowed the author to stay very close to the original design principles, rather than being forced into compromises to satisfy a schedule or other external requirements for steady progress.

We believe that this focus on very strong, explicit design principles, and a willingness to allow solutions to emerge

over time, rather than being forced into being, can help all language design processes avoid some of the camel humps of committee creations.

References

[1] S. Tucker Taft. The Design of ParaSail – Parallel Specification and Implementation Language. Web log. http://parasail-programming-language.blogspot.com September 2009.

[2] Honeywell-Bull. Ada Reference Manual. ANSI-MIL-STD 1815A, 1983.

[3] Barbara Liskov, Jeannette Wing. A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 16, Issue 6, November 1994.

[4] Dan Grossman et al. Region-Based Storage Management in Cyclone. *Programming Languages Design and Implementation 2002*. Berlin, Germany. June 2002.

[5] Intermetrics, Inc. Ada Reference Manual. ISO/IEC 8652:1995(E).

[6] Bjarne Stroustrup. The C++ Programming Language. Addison-Wesley, 1991.

[7] Bertrand Meyer. Object-Oriented Software Construction. Prentice-Hall, 1988.

[8] Bertrand Meyer. Introduction to the Theory of Programming Languages. Prentice-Hall, 1990.

[9] Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley, 1994.

[10] Bertrand Meyer. Eiffel, The Language. Prentice Hall Europe, 1992.

[11] Greg Nelson. Systems Programming with Modula 3. Prentice Hall, 1991.

[12] How Modula-3 Got Its Spots. http://www.modula3.org/threads/, issues 1 and 2.

[13] Barbara Liskov. A History of CLU. MIT-LCS TR-561, 1992.

[14] Ada Rapporteur Group. Annotated Ada Reference Manual. http://www.ada-auth.org/standards/95aarm/html/, 1995.

[15] S. Tucker Taft. The Ada 2005 Language Design Process. *CrossTalk* Vol 19, No. 8, August 2006.

[16] Niklaus Wirth. From Modula to Oberon. ftp://ftp.inf.ethz.ch/pub/software/Oberon/OberonV4/Docu/ModToOberon.ps.gz 1988.

[17] Andrew Koenig. C Traps and Pitfalls. Addison-Wesley, 1989.