Objection to Fast-track Ballot ECMA-372 in JTC1 N8037

I. Summary: N8037 contradicts an existing JTC1 standard

In response to document ISO/IEC JTC1 N8037, the UK objects to Fast Track Ballot ECMA-372 1st Edition C++/CLI Language Specification, on the grounds that there is a contradiction with an existing JTC1 standard. ISO/IEC 14882:2003 is the standard for the C++ programming language. Adopting a second standard under the proposed name of C++/CLI will cause unnecessary and harmful confusion in the marketplace.

We consider that C++/CLI is a new language with idioms and usage distinct from C++. Confusion between C++ and C++/CLI is already occurring and is damaging to both vendors and consumers.

A new language needs a new name. We therefore request that Ecma withdraw this document from fast-track voting and if they must re-submit it, do so under a name which will not conflict with Standard C++.

II. Arguments in support of the above points

Many of the following points are technical, to illustrate why we are concerned. If you are not expert in C++, or just not interested, feel free to skip this section.

CLI (Common Language Infrastructure), like Java, provides a virtual machine which interprets semi-compiled byte code and manages memory resources through garbage collection. Among its objectives are making it easier to combine modules written in different languages into a single application. It also aims to simplify the use of technologies such as XML, web services, and distributed programming.

At the time this project was launched in 2003, participants described it as an attempt to develop a "binding" of C++ to CLI, and a minimal (if still substantial) set of extensions to support that environment. C++/CLI is intended to be upwardly compatible with Standard C++, and Ecma TG5 have gone to praiseworthy efforts to guarantee that standard-conforming C++ code will compile and run correctly in this environment.

Nevertheless, **we believe C++/CLI has effectively evolved into a language which is almost, but not quite, entirely unlike C++ as we know it. Significant differences can be found in syntax, semantics, idioms, and underlying object model**. It is as if an architect said, "we're going to bind a new loft conversion on to your house, but first please replace your foundations, and make all your doors open the other way." **Continuing to identify both languages by the same name** (even though one includes an

all-too-often-dropped qualifier) **will cause widespread confusion and damage to the industry and the standard language.**

To list a few brief examples of these divergences:

C++/CLI adds more than two dozen new keywords to the 63 keywords of C++ syntax -- making the basic language more than a third larger.

Technically speaking, according to the letter of the C++/CLI document only 11 of these are unconditionally treated as keywords. Eight of those 11 keywords actually consist of two words (example: "enum class" -- two words which already have individual meanings in C++ code) separated by visually indistinguishable white space which is syntactically significant in binding them into one "keyword". An additional 13 are described as context sensitive identifiers, parsed as keywords only when they appear in certain places in the grammar. However, the online Microsoft documentation (http://msdn2.microsoft.com/en-us/library/xey702bw.aspx) refers to them unequivocally as keywords, and we are certain that few people will grasp the subtle difference laid out in the draft standard.

As a point of comparison, consider Java, a language with a heritage from C++ which is also interpreted at runtime by a virtual machine, which also uses garbage collection to manage memory resources, and which is also intended to be binary compatible across different execution environments. A quick overview shows that Java defines a mere 50 keywords, only a third of which are not shared with C++. An argument can be made therefore that on the evolutionary family tree of programming languages Java is a closer relative of C++ than is C++/CLI. And yet Java's authors did not choose to call it C++/VM, because they clearly consider it to be a different language.

C++/CLI uses some existing C++ keywords in new, non-standard constructs. One example is the keyword `new`, which can be used to indicate that a function is not an override of an inherited base class function of the same name. Another example is the standard access-specifiers `public`, `protected`, and `private` which C++/CLI uses in new constructs (to modify the definition of `class` or `struct`). It also adds several new access-specifiers to the grammar: `internal`, `protected public`, `public protected`, `private protected`, and `protected private`.

C++/CLI changes the meaning of currently valid C++ syntax. Perhaps a brief code example will illustrate some of our concerns:

```
// Example A -- this is C++
class Base {
    virtual void f1( int i );
    int f2();
};

class Derived : Base {
    int x;
```

```
  public:
    void f1( string s );
    int f2() { return x; };
};

// Example B -- this is C++/CLI
interface class Base {
    virtual void f1( int i );
    int f2();
};

ref class Derived : Base {
    int x;
  public:
    void f1( string s );
    int f2() { return x; };
};
```

In Example A, the member functions of Base are (by default) private, concrete functions, and f1() is virtual but not f2(). In Example B, the only visible code difference is the new keyword `interface`, but (by default) all member functions are public, abstract, and virtual (even if not identified as such).

In Example A, because Derived has the class-key `class`, not `struct`, it has private inheritance from Base (and therefore cannot be implicitly converted to Base). In example B, because Base is a CLI class and not a native C++ one, it confers public inheritance on Derived.

In C++, `Derived().f1( 42 );` would fail at compile time, because the declaration `Derived::f1( string );` hides the inherited function with the same name but different parameter. In C++/CLI, all inherited overloads are visible and callable; it is up to the programmer to beware whether she has unintentionally duplicated a name lurking much higher in the ancestry. *"[M]ember functions of native classes use hidebyname lookup ... On the other hand, member functions of ref classes, value classes, interface classes, and delegates use hidebysig lookup."* **C++/CLI adds considerable complication to the already complicated rules in Standard C++ for parsing and name lookup**

We feel that much of the functionality of C++/CLI could have been achieved without doing so much violence to standard syntax. The following is redundant but correct code in C++/CLI, and merely states explicitly what is now implied:

```
interface class Base {
  public:
    virtual void f1( int i ) = 0;
    virtual int f2() = 0;
};
```

```
ref class Derived : public Base {
    int x;
  public:
    using Base::f1;
    void f1( string s );
    int f2() { return x; };
};
```

Apart from the `interface` and `ref` keywords, the same class definitions are valid C++ syntax, and their meaning will not surprise a C++ programmer. To undermine the hard-learned expertise of millions of trained C++ programmers merely to save typing a few characters now and then (which in any case could be generated by an editor macro or CLI-aware IDE) is not justifiable.

C++/CLI changes the behavior of constructor and destructor functions. In Standard C++, an object has a deterministic lifetime, which begins and ends at well-defined places during the execution of the program. Necessary processing which must take place at beginning or end of an object's lifetime is placed in constructor and destructor member functions, so that it cannot be forgotten. This is one of the strongest benefits of using C++ instead of C or other languages, and many idioms of C++ rely on this behavior (often to ensure that files are properly closed, mutexes are unlocked, and scarce resources are freed).

In C++/CLI, calling a virtual member function within a constructor or destructor invokes a matching override from the most-derived descendant, which however may not be validly constructed when a base's constructor or destructor is running. In standard C++, virtual calls are shallow during construction and destruction.

In addition to destructors, C++/CLI classes can have finalizers, to be invoked by the garbage collector zero or more times at some non-deterministic time. In C++/CLI both finalizers and destructors of ref classes must be written so they can be executed multiple times and on objects that have not been fully constructed. It is difficult to describe the horror a C++ programmer feels at this change in the expected behavior of destructors.

C++/CLI adds a new way to create parameterised types and functions, collectively called "generics". **The angle-bracket syntax of generics is very similar to the Standard C++ mechanism of templates, but the semantics are different**, in that instantiation happens at runtime rather than at compile time.

C++/CLI syntax sometimes blurs the boundary between language features, library classes, and even coding conventions. The `for each` loop syntax (remember "`for each`" is a single keyword) only works on containers from the C++/CLI library which implement certain named member functions. Containers from the C++ Standard Library (which is included in C++/CLI by reference) do not implement these functions, and therefore will be seen as defective by programmers. This will be prejudicial to Standard C++ and will discourage its use.

C++/CLI introduces new idioms which conflict with long-standing idioms used in Standard C++ code. *"Although the* `explicit` *keyword is permitted on a constructor in a ref class or value class, it has no effect. Constructors in these classes are never used for conversions or casts"*. Instead, C++/CLI introduces static conversion functions, plus several new rules relating to the initialization of CLI types. Initialization is already one of the most complex issues in Standard C++; further complication is not welcome.

In C++/CLI, the `literal` and `initonly` keywords add functionality similar and in some cases equivalent to the use of `const`, which is also permitted in these contexts. The differences are subtle and confusing.

The semantics of `const` and `volatile` are changed. `const` (frequently used in C++ code) is only an optional modifier (modopt) in the CLI bindings, and therefore can be ignored by compilers and other tools, whereas `volatile` (rarely used in C++ code) is a required modifier (modreq). It is a standard idiom in C++ to make frequent use of the `const` keyword to prevent an object's value from being changed in unintended ways. This technique cannot be used with ref and value classes in C++/CLI.

A `#using` directive makes types from an assembly available in a source file. Despite its appearance, the document says `#using` is not a preprocessing directive. However, it does appear in the pre-processor section of the grammar in Annex A. This is just one of the ways in which **the C++/CLI preprocessor is incompatible with the preprocessor for Standard C++** (parsing the two-word keywords is another).

C++/CLI makes extensive use of garbage collection for allocated memory, a feature which is under discussion for Standard C++. But the approach adopted by C++/CLI has been considered and (at least tentatively) rejected as providing no benefit to existing source code. If Standard C++ adds a form of transparent garbage collection (instead of using a special kind of pointer type, like C++/CLI), the differences between the two languages will only grow larger.

There are other examples of ways in which the two languages can only grow further apart, when one compares decisions already taken or under debate in WG21 for C++0x with current C++/CLI and the future directions outlined in its annex.

III. Damaging confusion is already evident

Standard C++ is maintained by WG21, the largest and most active working group in SC22. WG21 meetings, twice a year lasting a week at a time, draw regular attendance by delegates from a number of national bodies and nearly all the important vendors of C++ compilers and libraries, plus a number of people who use the language in their work. By contrast, this Ecma draft was developed by a small handful of people -- awesomely competent ones, undoubtedly, but who do not represent the interests of the broad market of vendors and users. With ISO/IEC 14882 maintained by JTC 1 SC 22 WG 21 and C++/CLI maintained by ECMA, the differences between Standard C++ and the C++/CLI

variant will inevitably grow wider over time. The document proposes no mechanism for resolving future differences as these two versions of C++ evolve.

For JTC1 to sanction two standards called C++ for what are really two different languages would cause permanent confusion among employers and working programmers.

There is clear evidence that this confusion already exists now, even though C++/CLI has only recently been adopted as an Ecma standard and is commercially available from only one source.

Currently Microsoft is the only current vendor of a C++/CLI compiler, and also a leading vendor of a Standard C++ compiler. And yet **Microsoft's online documentation consistently confuses the syntax and semantics of C++/CLI with those of Standard C++.**

Documentation for Microsoft's Visual C++ product contains many code examples identified as "C++" -- *not* "C++/CLI" or even "C++.Net" -- which will fail to compile in a Standard C++ environment. See, for example, http://msdn.microsoft.com/visualc/default.aspx?pull=/library/en-us/dndotnet/html/NetFramework.asp, which has many examples showing parallel code for "C#", "Visual Basic", and "C++" (without the "/CLI" qualifier).

An article on "New C++ Language Features" at http://msdn2.microsoft.com/en-us/library/xey702bw.aspx contains this paragraph:

"*The following table lists new keywords that have been added to the C++ language. Note that some keywords consist of two words separated by white space.*" The page goes on to list what are officially context dependent identifiers, but it refers to them baldly as new keywords also, ignoring the subtle difference buried in the draft standard.

Note the statement that keywords have been "added to the C++ language" (no mention that it only applies to this new variant). There is no indication that using any of these new keywords renders code completely non-portable to other environments. Further pages with information on single keywords leave an even stronger impression that C++ is the language under discussion, not the C++/CLI extensions to it:

These pages consistently fail to distinguish clearly between Standard C++ syntax and extensions/adaptations for the CLI environment. Microsoft is not the only source of articles in which C++ and C++/CLI are considered equivalent, but they invented this new language and if they cannot tell them apart, it does not create confidence that average programmers will be able to maintain a clear idea of the many differences between them.

Possibly the largest faction of C++ programmers in the world are working mainly with Microsoft's compiler and platform. To them, Standard C++ is what Visual C++ does. If

they develop expectations that "C++" now has these new features, whether or not those expectations are accurate, **it will place at a disadvantage all competing products which offer conformance merely with Standard C++.** Portable programs will be disparaged for not taking advantage of platform-specific extensions available only in the Windows environment. (There is an open source implementation of CLI for other systems, called Mono, but to our knowledge currently no other compiler in the market supports C++/CLI.) The objective of using Standard C++ to achieve portable programs will be completely undermined.

C++ already has a reputation as a complicated language which is difficult to learn and use correctly. C++/CLI incorporates lip service support for Standard C++ but joins to it in shotgun marriage a complete second language, using new keywords, new syntax, variable semantics for current syntax, and a substantially different object model, plus a complicated set of rules for determining which language is in effect for any single line of source code.

If this incompatible language becomes an ISO/IEC standard under the name submitted, it will be publicly perceived that C++ has suddenly become about 50% more complex. The hugely increased intellectual effort would almost certainly result in many programmers abandoning the use of C++ completely.

Teaching materials, especially those which encourage the use of portable Standard C++, will have to be greatly expanded to counteract misconceptions created by the public perception that C++/CLI features have become a normal part of C++.

Future development of the standard language will be damaged, as there will be massive market resistance to adding any additional complexity on top, such as the changes planned for C++0x now undergoing development in WG21.

IV. A new language needs a new name

A parallel to this situation can be found in the history of C++ itself. As related by Bjarne Stroustrup in The Design and Evolution of C++, the language in its early days was known as "C with Classes", but he was asked to call it something else: *"The reason for the naming was that people had taken to calling C with Classes 'new C,' and then C. This abbreviation led to C being called 'plain C,' 'straight C,' and 'old C.' The last name, in particular, was considered insulting, so common courtesy and a desire to avoid confusion led me to look for a new name."*

In a spirit of helpful cooperation we even offer a few suggestions on possible new names: CLIpp or CliPP, CLI++ (the emphasis is on powerful access to CLI and more, since it supports unmanaged code too), or ++CLI (ditto, and also note that it contains "C++" spelled backwards), Ceeli (though this may conflict with an old ICL language for its mainframes), or even eCma++ (an obvious sister language to Ecmascript).

The UK request that Ecma withdraw this document from fast-track voting and if they must re-submit it to JTC1, do so under a name which does not include "C++".

This paper should not in any way be taken as suggesting that there is a sinister plot by Microsoft or anyone else to usurp or subvert the C++ Standard. Microsoft is an active participant in and a strong contributor to WG21. We accept that the people involved in this project are sincere in what they are trying to accomplish and do have persuasive (to them) reasons why they think these are good ideas. However, we also think they misunderstand what is important to other people.