Language Integration in the Common Language Runtime

Jennifer Hamilton
Common Language Runtime Development Team, Microsoft
jenh@microsoft.com

KEYWORDS
Language interoperability; common type system; exception handling; virtual machine; intermediate language; metadata.

ABSTRACT
The Common Language Runtime (CLR) is language and platform-neutral, and provides the underlying infrastructure for the Microsoft .NET Framework. A key innovation in the CLR is its support for multiple programming languages, enabling programming language integration at the runtime level to a much greater degree than is currently possible.

1. INTRODUCTION

The Common Language Runtime (CLR) provides the underlying infrastructure for the Microsoft .NET Framework [31]. It was designed with a number of goals in mind: to simplify the programming model, to provide a mechanism for simpler, safer application deployment and enable the development of rich web clients, to support scalability across a wide variety of systems and to provide a convergence point for a variety of different technologies, and to enable language integration far beyond simple language interoperability. In this paper, we will concentrate on the latter of these goals and describe the advancements made in the CLR with respect to supporting diverse programming languages and enabling language integration.

The CLR provides a language and platform neutral infrastructure that can support a wide variety of programming languages. This is achieved through a rich type system known as the Common Type System (CTS), and a language neutral intermediate language, Common Intermediate Language (CIL). The Common Type System supports a wide range of primitive types, including object references. The Common Language Specification (CLS), a subset of the CTS, defines a restricted set of types, still including object references, which can be manipulated by any CLS consumer. This model allows a compiler to target the CLR with its entire type system, and gain all of the advantages therein, but also defines a restricted set of types in order to ensure language/tool interoperability. CIL provides a language-neutral compiler target that can be input to a variety of code generators, in particular, just-in-time (JIT) compilers.

Attaining even minimal levels of language interoperability is quite difficult due to the wide variation in programming language types, features, and implementations. The CLR was designed to support object-oriented, procedural, and functional programming languages[1], generally in that order of importance. It currently supports a wide variety of the former two reasonably well, and provides minimal support for the latter. With respect to multi-language models however, the support provided for functional languages exceeds that currently available. Irrespective of language interoperability, the CLR uses metadata in conjunction with the type system and the intermediate language to completely describe a program and its type. The metadata is stored in the program executable along with the CIL and is processed through a metadata engine that provides APIs to both read and write the metadata. The use of metadata makes a CLR program self-describing, and allows other tools such as debuggers and profilers to query the type information.

We will begin with a brief discussion of platform and language interoperability, followed by a more detailed discussion of the CLR and how it has been designed to support language integration.

2. EXAMPLES OF PORTABILITY AND LANGUAGE INTEROPERABILITY

While many earlier programming languages were available on multiple platforms, the C programming language, which was designed to port the Unix operating system across multiple platforms, became one of the most successful attempts of source-level cross-platform portability. The language was reasonably simple, but supported a rich set of features. Coupled with a set of standard library functions, it became the language of choice for any form of system programming, and many other applications, across a multitude of platforms, particularly if portability was a concern. Most implementations also provided a

[1] Logic programming languages were not specifically considered, as [28] demonstrates that support for functional languages implies support for logic programming languages.

typically non-portable limited form of procedural language interoperation allowing, for example, C programs to call Fortran numerical libraries.

The C++ language grew directly from the C language, providing much richer language and class library features. This was achieved somewhat at the cost of platform portability, as many compilers didn't fully implement the newer features for quite some time, and very much at the cost of language interoperability beyond the procedural call level. Further, because the C++ standard did not specify the run-time model to be used by an implementation, programs running on the same platform but compiled with different C++ compilers typically cannot interoperate [15].

There have been several attempts to produce runtime systems that are language or platform independent. One such attempt, IBM's SAA [35] provided a common set of APIs across multiple platforms to provide both source code portability and language interoperability. However, in order to support such a wide variety of platforms, including MVS, OS/400, OS/2, and AIX, and languages, such as C, COBOL, Fortran, and RPG, the APIs were defined with a common denominator approach, reducing their usefulness.

A later attempt with somewhat similar goals, but with the addition of object-level binary compatibility, was IBM's System Object Model [8]. Rather than provide a specific API set, this model provided a language independent runtime through which languages could interoperate. SOM class interfaces were defined using the OMG CORBA [7] standard language called the Interface Definition Language, which is language-independent, although loosely based on the C++ language. The SOM IDL compiler generates language bindings for the target compiler (C or C++ were supported) and implementation languages corresponding to an IDL class definition. Bindings are language-specific macros and procedures that allow a programmer to interact with SOM through simplified syntax that is natural for the particular language. The bindings contain calls to the SOM runtime, which controls the instantiation, layout, and direct manipulation of classes and their instances. Alternatively, several compilers (for C++, Smalltalk, and OO-COBOL) provided DirectToSOM support [17], in which the compiler itself would either generate or consume IDL or generate calls directly to the SOM runtime. Using DirectToSOM, inter-language object sharing was achieved directly from the native programming language [16]. For a variety of reasons, some of which are discussed in [5], SOM did not achieve widespread acceptance.

At roughly the same time that IBM introduced SOM, Microsoft introduced COM [6]. Rather than define an API set or even a runtime model, COM is simply a specification for how conforming interfaces and instances must behave. An interface must support a specific minimal set of methods, such as `QueryInterface`, `AddRef`, and `Release`, and instances of the interface must consist of a vtable pointer that points to the list of supported methods. All methods must return an error number of a specified format (an `HRESULT`) that indicates the success or failure of the operation. The parameters passed on a method invocation must belong to a restricted set of mostly primitive types. This very simple model allowed diverse languages to interoperate just by being able to generate and consume COM interfaces. Although COM is criticized for its lack of true object-model support and the loss of fidelity at the interface [20], the simplicity of the model, which allowed it to be readily supported through a variety of compilers, contributed greatly to its widespread adoption.

The Java™ programming language is the most recent solution to the cross-platform portability problem. Java compilers generate a class file that conforms to a specific format as defined by the Java™ Virtual Machine Specification [24]. Compilers generate an intermediate language called Java™ bytecodes that are either interpreted or, more typically, JIT compiled to native code at runtime. A Java™ class file can be downloaded and executed on any machine that implements the virtual machine and the Java™ class libraries. Part of the virtual machine specification includes support to verify the type-safety of the program and ensure that no malicious behaviour is allowed. The explosion of the web and the desire to produce applications that can be downloaded to various computers in a safe manner greatly contributed to the widespread use of Java for developing cross-platform applications. However, Java™ severely impedes cross-language interoperability, for many of the same reasons as C++ before it, with additional restrictions such as garbage collected data which cannot be arbitrarily passed to other programs. While there are a variety of means available to communicate with non-Java™ programs [10], most suffer from additional runtime or programming overhead, or a loss of fidelity at the interface.

3. THE COMMON LANGUAGE RUNTIME

3.1 Design Goals

The Common Language Runtime (CLR) provides the underlying infrastructure for the Microsoft .NET Framework. It is platform-neutral, and consists of several components, including a garbage collector, class loader, metadata engine, and debugging and security services. The base class library in the .NET Framework provides the interface between the higher-level classes and the CLR. A major design goal of the CLR, and the focus of this paper, was to support a wide variety of

programming languages, in the spirit of COM, which would thereby enable language integration. Feedback and involvement from a variety of diverse language groups was solicited and incorporated throughout the design stage to achieve this goal [26]. As a result, there are many compilers that currently target the CLR, several of which are produced by Microsoft: C#[11], Java™, C++, and Visual Basic, and JScript. The remainder, including APL, COBOL, Component Pascal, Eiffel, Haskell# or Mondrian, Mercury, Oberon, Perl, Python, Scheme, and Standard ML, are produced by a variety of other companies and organizations [27].

All of the services, tools, and architecture information needed by compiler writers are fully documented as a published ECMA standard [12], with the intention that any organization can target the CLR. The class libraries provided with the .NET Framework can be accessed through any language that targets the CLR. Compiler writers can concentrate on the main task of parsing and generating code for their particular language rather than developing a runtime, which drastically reduces the barrier of entry. Further, any language can take advantage of the many tools and services provided with the CLR, such as profiling and security services and seamless, cross-language debugging. This is very compelling for producers of more marginal or esoteric programming languages as these languages would no longer be isolated, a factor which likely impedes their adoption [1]. With a compiler that targets the CLR, programmers using such languages would have direct access to the .NET Framework and any other applications written in any other language that targets the CLR. Further, their programs would be available for use by others in the same way. In addition, such programs could be deployed on the web, leveraging the type-safety and security support provided in the CLR [23].

3.2 Metadata Engine

With more traditional programming languages such as C++, once an application is compiled, there is very little information available about it that can be queried by outside tools. If there is such information, it is typically provided in a language-dependent format, such as Java™ class files, or a proprietary format, such as Visual C++ PDB files. COM type libraries do provide information in a language-independent format [6], but have other drawbacks in that they are difficult to extend and the APIs are C++-specific. The CLR Metadata Engine was designed to provide a language-independent schema and supporting APIS to enable the CLR development model. When a program is compiled with a CLR-enabled tool, information describing the classes defined and used in that program are emitted as metadata through the Metadata Engine and stored with the resulting program object along with the CIL.

The basic form of metadata extensibility is through *custom attributes*, which are arbitrary types (rooted at `System.Attribute`) that can be associated with the object. The metadata engine provides support to then enumerate and retrieve custom attributes at runtime. This provides an extensible and type-safe way to extend the metadata, which contributes to the ability of the CLR to support a broad range of languages. For example, the metadata does not directly support the concept of `const` methods, but a C++ compiler can generate a custom attribute for a method to represent this modifier and can then query for it when compiling a reference to that particular method to ensure consistency. Tools that do not recognize the attribute can ignore it. As we will discuss later with the Common Language Subset, however, such an attribute would likely not be included in a public interface.

CLR programs are self-describing in that the metadata includes all type and method information necessary to completely describe types, methods and their parameters. The metadata information can then be queried by other tools, such as the debugger and profiler through a native COM interface, or through .NET reflection framework. The CLR uses the metadata extensively for a variety of tasks, including object instantiation, type safety verification, and JIT compiling the CIL to native code. The purpose of the paper is not to describe the CLR in its entirety, but rather those features that are specific to language support and integration, so the metatdata format will not be described in detail (see [12] for further information).

3.3 Common Intermediate Language

Compilers that target the CLR generate Common Intermediate Language (CIL), rather than native code, storing the CIL directly into an executable file[2]. When the file is subsequently loaded for execution, the CIL will be compiled to native code by a just-in-time compiler as required. There is also an option to JIT the entire file prior to execution, which improves runtime performance in that the methods are already JIT compiled. However, the code generated is not quite as optimized as the standard JIT compiler, which can take into account the current execution environment.

CIL is a stack-based, type-neutral intermediate form. It was designed for compilation a JIT compiler and never an interpreter, so there is no type information contained in the instruction stream. All type information is obtained through token references

[2] The C++ compiler can generate native code directly, but that is beyond the scope of this discussion.

to metadata artifacts. For purposes of brevity, the paper does not go into further detail about the specifics of CIL, except where it relates to language support and integration issues. Further details about CIL can be found at [12].

4. SUPPORTING MULTIPLE LANGUAGES

There are currently a large number of language projects that successfully target the Java™ VM with varying degrees of success [4], [32]. This raises the question of how the CLR contributes to multiple language support beyond this baseline. The Java™ was designed specifically to support the Java™ programming language, and as such it supports only features that exist in that language. There are a variety of features required by other languages that are not available, some of which can be implemented with a loss of efficiency to varying degrees, and others that simply cannot be supported. Examples of such features that have been included in the CLR specifically to enable multiple programming languages are:

Unsafe code There are several features supported by the CLR that are classified as *unsafe* [23]. Support for unsafe features is closely tied to the security system and functions using such features must run in a "fully-trusted" environment. Supporting unsafe code allows programmers to perform unverifiable operations without leaving the CLR environment and enables support for languages that require inherently unsafe features. For example, the CLR-enabled C++ compiler (known as "C++ with Managed Extensions" where the term "managed" refers to the fact that the CLR is managing the execution environment) cannot generate verifiable code, due to the number of C++ constructs that prohibit this. Therefore all managed C++ programs must run in a fully-trusted environment.

Tail calls Languages such as Scheme require implementations to be properly tail-recursive, so that the execution of an iterative computation can take place in constant space [13]. However, because the JVM requires that any implementation encode control-flow for stacks as heap-allocated objects, a constant-space implementation cannot be achieved. Limited tail-recursion can be supported, but general tail call support cannot be implemented in a portable way [1]. The CIL definition includes a tail call prefix to the **call** instruction which causes the stack frame of the current method to be released before control is transferred to the callee.

User-defined value types Many programming languages, such as Visual Basic, COBOL, Fortran, and C++, require the ability to declare and use data on the program stack rather than through the garbage-collected heap for efficiency reasons. The CLR supports user-defined value types to allow arbitrary types to be defined that will be allocated on the stack.

Enumerations The CLR supports typed enumerations whereby a type is declared whose instances can take on a value from, and only from, a set of named constants.

Unsigned integers The CLR supports 8, 16, 32 and 64-bit unsigned integers.

Variable length argument lists The CLR support for variable length argument lists ensures type-safety of the arguments passed.

Pointer arithmetic To support languages such as C++, full pointer arithmetic support is a necessity. This is an example of an *unsafe* feature discussed earlier. Any function that contains pointer arithmetic instructions will be treated as unsafe.

Global variables The CLR supports the definitions of variables that are globally accessible throughout the program and are not related to a specific class definition.

Private scope Languages such as C++ have the concept of a variable with private scope, which is globally accessible within the compilation unit, but inaccessible beyond it.

Call indirect The CLR supports function pointers, which statically capture the address of a given function and can be subsequently used to invoke the function via the captured address.

By-reference parameters The CLR allows parameters to be passed by-reference, so that updates to an argument in a called method will be reflected in the caller's state.

Multidimensional arrays The CLR supports true multidimensional arrays, rather than implementing them as arrays of arrays.

Pinned objects In order to allow a garbage-collected object to be passed outside the CLR environment to native code (an unsafe feature), the CLR provides the ability to *pin* an object so that it will not be moved if a garbage collection occurs[3]. For example, a managed C++ application can obtain the address of the string buffer corresponding to a heap object and pass this address to a native function.

5. COMMON TYPE SYSTEM

This section provides a brief overview of the Common Type System (CTS), which is the formal type system implemented by the CLR. A type in CTS refers to both the interface and the implementation. While the CTS does include the concept of an interface declaration that does not and cannot contain an implementation, types and type inheritance always refer to both. A type may be marked as *abstract*, in which case all methods need not be defined, but otherwise a type provides a complete implementation of all methods defined therein.

[3] There is a performance overhead in pinning an object, as the garbage collector has to manage such objects separately.

There are two basic types supported: value and reference types. A value type is used for representing bit sequence values such as integers and is allocated on the stack instead of the garbage-collected heap. The value type describes the representation of the underlying storage and what operations are valid, but the type is not stored with the instance, so the type must be known to correctly manipulate the instance. A reference type carries more information than a value type. Instances of a reference type have a unique identity and contain a handle to their type, so it is always possible to determine the exact type of any reference instance. Any value type instance can also be *boxed*, which creates a corresponding reference type instance. A boxed type can always be *unboxed*, which recreates the original value type instance.

Unboxed value types have no parent and all other types must inherit from exactly one other type. The type hierarchy is rooted at the class `System.Object` from which all other types ultimately inherit. Boxed value types inherit only from `System.ValueType` or `System.Enum` and cannot be further refined. Other types may be declared as *sealed* to indicate that no further derivation is allowed. While the CTS supports only single type inheritance, it does support multiple *interface* inheritance. Interfaces do not have an associated implementation for instance methods, but static methods can be defined. Any complete type that inherits from one or more interfaces must provide an implementation for all methods declared in the inherited interfaces.

In order to support a wide variety of languages with optimal runtime efficiency, there are a number of built-in value types defined by the CTS that are supported directly by the CLR. These types are shown in Table 1 (the types marked with an √ are part of the *Common Language Specification*, which will be discussed in the next section). CTS types may be defined with a variety of members, including fields, methods, properties and events. Fields and methods may be static, in which case they are associated with the type rather than a particular instance of the type. Methods may be virtual or non-virtual, and virtual methods may be marked as *final*, indicating that they cannot be overridden by a derived class method. Virtual methods may also be marked as *new*, indicating that they are placed in a new virtual function table slot, separate from any matching parent methods. A virtual method marked as *abstract* does not have an associated implementation, but can only be defined in a type that is also marked abstract. All interface methods must be marked as virtual.

Table 1. CTS Built-In Types

CTS Name	Description
√ bool	True/false value
√ char	Unicode 16-bit char.
√ class System.Object	Object or boxed value type
√ class System.String	Unicode string
√ float32	IEEE 32-bit float
√ float64	IEEE 64-bit float
int8	Signed 8-bit integer
√ int16	Signed 16-bit integer
√ int32	Signed 32-bit integer
√ int64	Signed 64-bit integer
native int	Signed integer, native size
native unsigned int	Unsigned integer, native size
typedref	Pointer plus runtime type
√ unsigned int8	Unsigned 8-bit integer
unsigned int16	Unsigned 16-bit integer
unsigned int32	Unsigned 32-bit integer
unsigned int64	Unsigned 64-bit integer

Properties are named values with associated methods to manipulate the value. By convention, a getter is supplied to retrieve the value and an optional setter to modify the value, but there no restrictions placed on the methods associated with a property, either in name or usage. Events provide a mechanism for registering and deregistering for an event notification and issuing one. They are supported in much the same way as properties, with different conventional named methods.

Members can be defined with one of seven accessibility rules: private scope, private, family, assembly, family and assembly, family or assembly, and public. A member with *family* accessibility can be accessed only from a derived type, while a

member with *assembly* accessibility can be accessed from any type within the given types' assembly. An assembly is a defined collection of application units that provides name scoping and class versioning, among other things.

Types may be generated statically by a tool or compiler, or dynamically at runtime through the *reflection* framework. This framework supports the complete definition of a type and its associated members at runtime, including generating CIL for member definitions. The reflection framework may also be used to query type information at runtime, or perform late-bound method invocation. The CTS defines the behaviour of most aspects of the object model, but does not define how method overloading is handled. This is an intentional omission. Because each language has different rules about method overloading, it is always resolved by compiler, which specifies explicitly how the CLR should treat each method, either as virtual, non-virtual, or new.

By default, the CLR controls the layout of type instances, and this layout is not specified. However, in order to support various language features, a type may be defined as having either *sequential* or *explicit* layout. Sequential layout instructs the CLR to layout the instance fields in the order of definition within the type but with no other restriction on a field's specific offset (to allow for alignment or padding), however it is possible to provide an explicit alignment boundary for fields. Explicit layout allows the generating tool to indicate explicitly the offset of each field within the type instance. A tool can also specify an overall size for a type, which is typically used for value types, where the underlying field layout will be controlled by the tool and is unknown to the CLR.

6. COMMON LANGUAGE SPECIFICATION

A common type system is fundamental to language integration. Incompatible types are the primary barriers that keep languages from interoperating. With COM and other such models, each language targeting the interface must usually convert from internal types to interface types. For example, structures and member data cannot be passed through the COM interface; only specific primitive types are allowed. In order to allow a wide variety of languages to target the CLR, but to also have a guarantee of language integration, the Common Language Specification (CLS) defines a subset of the CTS that each CLS-compliant *consumer* or *extender* must support. A CLS-compliant consumer is a tool that can use any CLS-compliant type. A CLS-compliant extender can use or generate any CLS-compliant artifact. As an example, the C# compiler is a CLS-compliant extender as it can both generate and consume CLS-compliant artifacts, while the CLR debugger is just a consumer, as it cannot generate new artifacts.

The CLS is defined through a set of forty rules succinct rules designed to provide a useful subset of the CTS that can readily be implemented in most languages. This section highlights the major restrictions and features of the CLS subset. Note that the restrictions apply to non-private types and members. Private entities that are not visible to a consumer have no such restrictions. The CTS imposes only two restrictions on names: they must be encoded as 16-bit Unicode strings and they cannot contain an embedded NULL. The CLS imposes further restrictions to ensure language integration, one being that visible distinct names cannot differ only in case so that case-insensitive languages, such as Visual Basic and COBOL, are able to participate. All names used in a scope must also be unique, independent of their kind. For example, a field and a method in the same scope may not have the same name. Languages must provide a means of referencing identifiers that collide with keywords in the language. In addition, languages that extend the CTS must allow the use of language keywords for defining and overriding of virtual methods.

A CLS-compliant interface can define only virtual methods, properties, and events. Static and instance methods and fields are not allowed. The CLS defines specific naming rule for properties and events. The CLS supports a restricted set of predefined types. Those types shown with a '√' in Table 1 are included in the CLS, the remainder are not. A conforming CLS type or signature may include only those predefined types that are part of the CLS, or user-defined types that are also CLS-compliant. Arbitrary boxed value types are not part of the CLS. For each CLS predefined type, there is explicitly defined a corresponding framework value type, such as `System.Boolean` for `bool` and `System.Double` for `float64`. A CLS-compliant artifact may only reference other CLS-compliant artifacts. For example, a method parameter in a CLS-compliant type cannot be `int8`. An artifact is designated CLS-compliant using the `System.CLSCompliantAttribute`. This allows tools to specify compliance and also to determine whether or not a referenced entity is compliant when defining a CLS-compliant signature or type.

An application is not required to make use of the Common Language Subset. The intention of the CLS is to provide a common subset through which diverse languages can interoperate, but if language interoperability is not a consideration, then the CLS can be safely ignored. However, if one were providing an interface to a component that may be used by other languages, that interface should be made CLS-compliant. This would not restrict the entire application to being CLS-

compliant, only the exposed interfaces. For example, C# is a case-sensitive language. One could write a component in C# and not be concerned with the case-insensitivity rules of CLS-compliance. If any interface of that component were to be made CLS-compliant, this would require that particular interface's signature to be compliant with respect to case-insensitivity and the types used in the interface. But beyond the actual exposed interface, the remainder of the application can remain incompliant and take advantage of any features available in the CLR.

7. EXCEPTION HANDLING

Error handling can be somewhat error-prone if exception-handling support is not available. For example, when using COM, errors are based on a return code known as an HRESULT. These return codes can be quite cryptic and difficult to debug. Further, the major flaw with any return code-based model is that if the caller does not explicitly check for the return code, an error can go unnoticed. This can also lead to resource leaks in error conditions without very careful programming.

Error handling in the CLR is done exclusively through exceptions, using a two-pass run-time stack unwinding model [33], where the first pass searches for a handler for the exception and the second pass performs any cleanup before invoking the target handler. All languages targeting the CLR must support or tolerate exceptions. Exception information is specified to the CLR in the form of an exception clause table that precedes the CIL for the method. Each exception clause takes the following form:

Discriminator
Try block offset
Try block length
Handler block offset
Handler block length
Class token or Filter block offset

The discriminator indicates if the handler is a *typed* or *filter-based* exception handler, or *termination*, handler. Each handler guards a specific code range, as given by the try block offset and length. All offsets and lengths are with respect to the IL instruction stream for the function. A typed exception handler is called when the thrown exception instance matches (either directly or as a super-type of) the specified type exception. For such handlers, a class token representing the exception type to be caught is provided. The handler is specified with the handler offset and length. A filter-based exception handler consists of a filter expression, which is evaluated during the first pass, and a handler that is called on the based on the value of the filter expression. The filter expression can evaluate the exception information at runtime to determine if the handler should be called. For filter-based handlers, the offset to a filter expression is supplied instead of a class token. Termination handlers are called during the second pass to perform any cleanup prior to passing control to the handler.

The CLR uses a late bound approach in that no work is done until an exception occurs, at which point the exception handling support will search for a method to handles the given exception. Processing begins on the first pass with the current method on the stack. Each exception clause, if any, is examined in the order that the clauses are specified in the table. If the clause defines an exception (as opposed to a termination) handler and the current execution point within that function falls within the range guarded by the handler, the class token or filter expression is evaluated to determine if the handler will catch the exception. This continues with each clause in the table and each subsequent function on the call stack until a handler is found, which completes the first pass. When a handler is found, the second pass is performed, which invokes any termination handlers found between the point of the exception and the catch point that guard active code on the call-stack. As with exception handlers, termination handlers are invoked in the order specified in the table.

Exception handling is a particularly difficult, but extremely important, aspect of language integration to implement, particularly for languages that naturally rely on exceptions for error handling. None of the attempts to provide language interoperability discussed earlier provided any mechanism for cross-language exception handling, whereby an exception can be raised on one language and caught in another. The C-- portable assembly language supports multiple exception models [33], but not within the context of language interoperability.

8. TARGETING THE CLR

When a programming language is targeted to the CLR, there are two areas of interest for the compiler writer. The first is how to represent that language in the CLR. Many language features map naturally to some aspect of the CLR, however, some are less obvious or not possible. For example, the CLR does not support nested functions, so the Mercury implementation handled this by hoisting such functions to the top-level through transformations. A discussion of the experiences in porting Mercury and Oberon to the CLR can be found in [9] and [14] respectively.

The second issue to be addressed by a compiler writer is how various features of the CLR might be manifested in the language, in particular the minimally-required features of the Common Type System that may not be defined in the language. As an example, consider static methods, instance constructors, and static fields. With the C++ implementation, these are exposed as you would expect for that language. By contrast, with the SML.NET implementation [34], static methods are exposed as top-level function bindings, instance constructors as functions with the same name as their containing class, and static fields as top-level value bindings. These did not require enhancing the language, but merely choosing how to expose CLR features through existing language concepts (or vice-versa). In order to avoid the perils of a common-denominator approach to language integration, there are a variety of features that the Common Language Subset dictates that may not be available in all programming languages, such as inheritance and exception handling. For example, the Visual Basic programming language had no concept of inheritance prior to targeting the CLR. Support for inheritance was added specifically to enable the language to the platform. As another example, class types with virtual methods could not be represented without language extensions to SML, as it is not an object-oriented language. Extensions similar to those described in [2] were required.

9. RELATED WORK

Earlier in the paper we discussed other examples of language interoperability and multiple language support. There are certainly additional examples of language interoperability support, but restricting the discussion to object-oriented or component-base models, SOM, COM, and the languages targeting the Java™ VM are the only major recent examples.

Of the three, SOM provides the greatest level of language integration, allowing diverse languages such as Smalltalk and C++ to share objects [16]. However, SOM is not a virtual machine-based model and provides no machine-independent layer through which type-safety verification and other features might be achieved. Compilers supporting SOM generate native code with calls to the SOM runtime. The COM model is even simpler, providing basically a set of rules and restricted types through which languages can generate and consume interfaces [6].

The Java™ was designed specifically to support the Java™ programming language, and as such it supports only features that exist in that language. While there are currently a large number of language projects that successfully target the Java™ VM with varying degrees of success [32], the goal of such projects has not been general language interoperability (beyond Java™ and the language targeting the VM), but rather to take advantage of the features of the Java™ VM to enable that particular programming language. There are a variety of features, which we discussed earlier, required by other languages that are not available, some of which can be implemented with a loss of efficiency to varying degrees, and others that simply cannot be supported.

The G- portable assembly language [21] supports multiple programming languages, but is not designed to support integration of those languages, but rather is designed to support the generation of high-quality machine language on a variety of platforms through a platform-neutral target language. The Thor object-oriented database [25] defines a language-independent database interface language, but the problem domain is restricted to that of data persistence, not of in-memory sharing of object state. Likewise, the Typed Assembly Language (TAL) [29] provides a language-neutral assembly language more suitable than Java™ bytecodes as a target for heterogeneous languages, but the scope is limited to the definition of an intermediate language.

The CLR is unique in that it is a virtual machine-based model, providing type-safety and security, that supports most features required by traditional programming languages and many of the features required by more esoteric languages. As discussed earlier, there are currently a wide variety of programming languages that successfully target the CLR, ranging from C++ to Mercury to COBOL. By defining the CLS and providing compile-time and run-time support to enforce it, while allowing a component to also make use of the full range of non-CLS compliant features, the CLR is able to avoid the common-denominator approach inherent in some of the language interoperability models discussed earlier.

10. LIMITATIONS AND FUTURE WORK

There are a several language features that were not included in the initial release of the CLR. Multiple inheritance was specifically omitted due to the differences in various language implementations; there was no efficient way to implement this feature within the context of dynamic class loading. *Call-with-current-continuation,* which allows non-local exits to be performed from loops or procedures in a structured way [13], was not included partly for the same reason, but also because structured exception handling provides a better way to achieve much of the same functionality.

There are several features that were omitted due to time constraints, but will likely be supported in a future version. These include generics and structural type equivalence. Rather than support generics as a compile-time feature, as do languages such as C++, generics would likely be supported directly by the CLR, which is possible due to the type-neutral aspect of CIL [22]. As with the Java™ VM, array assignment is in the CLR is covariant, therefore runtime checks must be made to ensure any update operations do not violate the type system [3], [30]. The availability of generics could alleviate this overhead.

With structural type equivalence, two types are considered equal if their underlying structure is the same, as opposed to name equivalence, where the names must match. Supporting structural equivalence would also simplify the implementation in functional programming languages of *closures*, a function body together with an associated environment [18], and *currying*, where a function of N arguments may be implemented as a function of one argument which returns another function of N-1 arguments [19]. These and other potential extensions to the CLR to better support functional languages are described in [36].

11. CONCLUSION

The multi-language support and language integration capabilities of the CLR are a major advancement in programming language implementation. Through a rich type system and a language neutral intermediate language, the CLR provides a language and platform neutral infrastructure that can has been targeted by a wide variety of programming languages. It currently supports many object-oriented and procedural languages reasonably well, and provides minimal support for several functional programming languages. By targeting the CLR, compiler writers can concentrate on the main task of parsing and generating code for their particular language rather than developing a runtime, which drastically reduces the barrier of entry. Programs generated by any compiler that targets the CLR have access to the entire .NET Framework and to any other applications written in any other compliant language. This is very compelling for producers of more marginal or esoteric programming languages as these languages would no longer be isolated, a factor which likely impedes their adoption.

12. ACKNOWLEDGEMENTS

This project is the culmination of several years of work by the members of the Common Language Runtime team, other teams at Microsoft, and various organizations outside the company. It would be infeasible to list all the names, but suffice it to say that there were many key contributors to this project.

13. REFERENCES

[1] Benton, N., A. Kennedy, and G. Russell, "Compiling Standard ML to Java Bytecodes", *Proceedings of the 3rd ACM SIGPLAN Conference on Functional Programming*, September 1998.

[2] Benton, N., and A. Kennedy, "Interlanguage Working Without Tears: Blending SML with Java", *ACM Conference on Functional Programming*, September, 1999.

[3] Benton, N. Private correspondence, November 2000.

[4] Bothner, P. "Byte-compilation of Scheme using Java byte-codes", http://www.mit.edu/afs/sipb/project/kawa/doc/scm2java.html.

[5] Box, D. "The Component Object Model and Some Other Model: A comparison of technologies revisited yet again", unpublished paper.

[6] Box, D. *Essential COM*, Readingham, MA: Addison-Wesley, 1998.

[7] *The Common Object Request Broker: Architecture and Specification*. Farmingham, MA: Object Management Group, 1992

[8] Danforth, S., P. Koenen, and B. Tate, *Objects for OS/2*. New York: Van Nostrand Reinhold, 1994.

[9] Dowd, T., F. Henderson, and P. Ross, "Compiling Mercury to the .NET Common Language Runtime" in *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, September 2001.

[10] Eckel, B. "Appendix A: Using non-Java Code", *Thinking in Java*, http://www.ldeo.columbia.edu/~cdong/AppendixA.html

[11] *ECMA-334 C# Language Specification*, ECMA, December 2001.

[12] *ECMA-335 Common Language Infrastructure (CLI)*, ECMA, December 2001.

[13] Kelsey, R., W. Clinger, and J. Rees. "Revised Report on the Algorithmic Language Scheme", *ACM SIGPLAN Notices, September 1998*. With H. Abelson, N. Adams, D. Bartley, G. Brooks, R. Dybvig, D. Friedman, R. Halstead, C. Hanson, C. Haynes, E. Kohlbecker, D. Oxley, K. Pitman, G. Rozas, G. Steele, G. Sussman, and M. Wand.

[14] Gutknecht, J. "Active Oberon for .NET: An Exercise in Object Model Mapping" in *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interoperability*, September 2001.

[15] Hamilton, J. "Reusing Binary Objects with DirectToSOM C++", *C++ Report*, March 1996.

[16] Hamilton, J. "Interlanguage Object Sharing with SOM", *Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies*, 1996.

[17] Hamilton, J. *Programming with DirectToSOM C++*. New York: John Wiley and Sons, 1996.

[18] Harper, R. "Functionals", *Programming in Standard ML* http://www.cs.cmu.edu/afs/cs.cmu.edu/usr/rwh/public/www/introsml/core/functionals.htm

[19] Hudak, P., J. Peterson, and J. Fasel, "Functions", *A Gentle Introduction To Haskell*, http://www.haskell.org/tutorial/functions.html

[20] IBM, "The System Object Model (SOM) and the Component Object Model (COM): A comparison of technologies from a developer's perspective", http://www-4.ibm.com/software/ad/som/library/somvscom.html

[21] Jones, S., Ramsey, N., and F.Reig, "C--: a portable assembly language that supports garbage collection", http://www.cminusminus.org/abstracts/ppdp.html

[22] Kennedy, A. and D. Syme, "Design and Implementation of Generics for the .NET Common Language Runtime", *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*.

[23] LaMacchia, B., S. Lange, M. Lyons, R. Martin and K. Price, *.NET Framework Security*. Addison Wesley Professional, 2002.

[24] Lindholm, T. and F. Yellin, *The Java Virtual Machine Specification*, Reading, Mass: Addison-Wesley, 1994.

[25] Liskov, B., M. Day, S. Ghemawat, R. Gruber, U. Maheshwari, A. Myers, and L. Shrira. "The Language-Independent Interface of the Thor Persistent Object System" in *Object-Oriented Multidatabase Systems* (O. Bukhres and A. Elmagarmid, eds.) Prentice-Hall, 1994.

[26] "Mercury on Microsoft's .NET Framework", http://www.cs.mu.oz.au/research/mercury/information/dotnet/mercury_and_dotnet.htmls

[27] Microsoft .NET Language Partners, http://www.msdn.Microsoft Corporation.com/vstudio/partners/language/default.asp

[28] Miller, J. "MultiScheme: A Parallel Processing System Based on MIT Scheme", MIT LCS/TR/402

[29] Morrisett, G., K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "TALx86: A Realistic Typed Assembly Language", *ACM SIGPLAN Workshop on Compiler Support for System Software*, 1999.

[30] Myers,. A, J. Bank, B. Liskov, "Parameterized Types for Java", *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1997.

[31] .NET Developer Center, http://www.msdn.microsoft.com/net/

[32] "Programming Languages for the Java Virtual Machine", http://grunge.cs.tuberlin.de/~tolk/vmlanguages.html

[33] Ramsey, N., and S. Jones, "A Single Intermediate Language that Supports Multiple Implementations of Exceptions", *ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation.*

[34] Standard ML, http://research.microsoft.com/Projects/SML.NET/

[35] *Systems Application Architecture: An Overview*. IBM GC26-4341

[36] Syme, D. "ILX: Extending the .NET Common IL for Functional Language Interoperability", in *BABEL'01: First International Workshop on Multi-Language Infrastructure and Interopera bility*, September 2001.