Miller-CHP1.fm Page 1 Wednesday, September 24, 2003 1:50 PM

1. Introduction to the Common Language Infrastructure

The Common Language Infrastructure (CLI) is an International Standard that is the basis for creating execution and development environments in which languages and libraries work together seamlessly. The CLI specifies a *virtual execution system* that insulates CLI-compliant programs from the underlying operating system. Where virtual execution systems are developed for different operating systems, programs written with CLI-compliant languages can be run in these different systems without recompiling, or worse, rewriting.

Programming with CLI-compliant languages ultimately gives the programmer a simple but rich development model, allowing development in multiple languages, promoting code reuse across languages, and removing most of the plumbing required in traditional programming. The CLI makes it possible for modules to be self-registering, to run in remote processes, to handle versioning, to deal with errors through exception handling, and more.

This book, by amplifying the standard, provides a blueprint for creating the infrastructure for this simpler programming model across languages and across platforms. Because the theme of the CLI is broad reach, it also includes provisions for running modules compiled by existing languages into "native code"—machine code targeted at a specific system. This is called **unmanaged code**, as opposed to the managed code that is CLI-compliant.

This book also describes what is required of languages to be CLI-compliant, and what library developers need to do to ensure that their libraries are accessible to any programmer writing in any CLI-compliant language. In addition, it provides the guidelines for implementing a virtual execution system, which insulates executables from the underlying operating system.

This chapter is an overview of the CLI and attempts to provide a key to understanding the standard. In addition, throughout the specification, annotations explain many of the details—either clarifying what is written in the specification or explaining the origins of some of its elements.

Miller-CHP1.fm Page 2 Wednesday, September 24, 2003 1:50 PM

Components of the CLI

It is most important at the beginning to understand the basic elements of the CLI and how they are related. These elements are

- Common Language Infrastructure (CLI)
- Common Type System (CTS)
- Common Language Specification (CLS)
- Virtual Execution System (VES), which executes managed code and lies between the code and the native operating system

Figure 1-1 shows how these elements are related. Together, these aspects of the CLI form a unifying specification for designing, developing, deploying, and executing distributed components and applications.

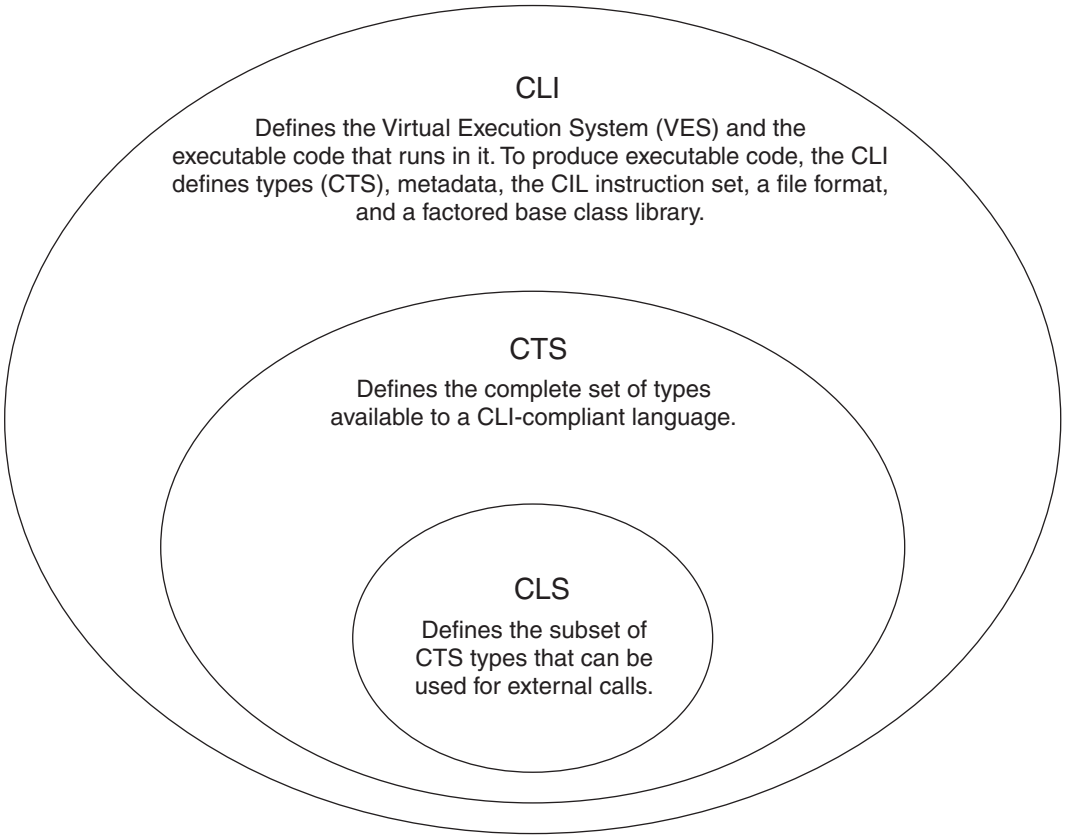

Figure 1-1 Relationship of the CLI, the CTS, and the CLS

Miller-CHP1.fm Page 3 Wednesday, September 24, 2003 1:50 PM

Each programming language that complies with the CLI uses a subset of the Common Type System that is appropriate for that language. Language-based tools communicate with each other and with the Virtual Execution System using metadata to define and reference the types used to construct the application. When a constructor is called to create an instance of an object, the VES uses the metadata to create instances of types, and to provide data type information to other parts of the infrastructure (remoting services, assembly downloading, security, etc.).

Languages and programming environments that do target the CLI—there are currently more than 20, and the list is growing—produce what is called **managed code** and **managed data**. The key to these is **metadata**—information associated with the code and data that describes the data, identifies the locations of references to objects, and gives the VES enough information to handle most of the overhead associated with older programming models. This overhead includes handling exceptions and security, and providing information to tools that can ensure memory safety. It may also include running on remote systems by creating proxies for the programmer, as well as managing object lifetime (called **garbage collection**).

Among the things the CLI specifies are the following:

- The Common Type System
- The Common Language Specification for publicly available calls
- Metadata
- Portable file format for managed code
- The Common Intermediate Language (CIL) instruction set
- Basic requirements of a Virtual Execution System
- A programming framework built on top of all of this

The CLI also bridges the managed and unmanaged worlds. The CLI describes how, in the same program, managed modules can be run with unmanaged modules compiled in native code (machine code, specific to a given system). This interoperation is also crucial to describing how modules can communicate though the VES with the underlying operating systems.

The Common Type System

Ultimately, all programs are built from data types. At the core of every language are built-in data types, ways of combining them to form new types, and ways of naming the new types so that they can be used like the built-in types.

Miller-CHP1.fm Page 4 Wednesday, September 24, 2003 1:50 PM

Data types are more than just the contents of the bits that the data occupy. They are also the methods that can be used to manipulate them. In value-oriented programming, "type" usually means data representation. In object-oriented programming, it usually refers to behavior rather than to representation. The CTS combines these notions, so "type" means both of these things: two entities have the same type if and only if they have both compatible representations and compatible behaviors. Thus, in the CTS, if a type is derived from a base type, instances of the derived type may be substituted for instances of the base type because both the representation and the behavior should be compatible.

The idea of the Common Type System is that compatible types allow language interoperation. If you can read the contract provided by any type and use its operations, you can build data structures and use your control structures to manipulate them.

The CTS presents a set of rules for types. As long as you follow those rules, you can define as many of those types as you like—in effect, the types are extensible, but the type system is not. For example, you can define any object or value type you like, as long as it follows the rules, but you cannot, for example, define a CTS-compliant type that uses multiple inheritance, which is outside of the type system.

The Common Type System was designed for broad reach: for object-oriented, procedural, and functional languages, generally in that order. It provides a rich set of types and operations. Although many languages have types that they have found useful that are not in the CTS, the advantages of language integration usually outweigh the disadvantages. Out of 20 languages that carefully investigated the CTS, at the time of this writing 15 have chosen to implement it.

The CTS is described in detail in section 8 of Partition I (Chapter 2 of this book).

The Common Language Specification

The Common Language Specification (CLS) is a subset of the Common Type System (CTS). It is a set of types that may be used in external calls in code that is intended to be portable. All of the standardized framework (described in Partition IV, including the Base Class Library, XML Library, Network Library, Reflection Library, and Extended Numerics Library) are intended to be used on any system running a compliant VES, and in any CLS-compliant language. Therefore, the framework follows the CLS rules, and all (well, almost all) the types it defines are CLS-compliant to ensure the broadest possible use. In the few cases in which types or methods are not CLS-compliant, they are labeled as such (that's one of the CLS rules), and they are intended for use by compilers and language runtimes rather than direct use by programmers.

Figure 1-2 shows the relationship of languages, the Common Type System, and the Common Language Specification. It illustrates examples of how two compliant languages,

Miller-CHP1.fm Page 5 Wednesday, September 24, 2003 1:50 PM

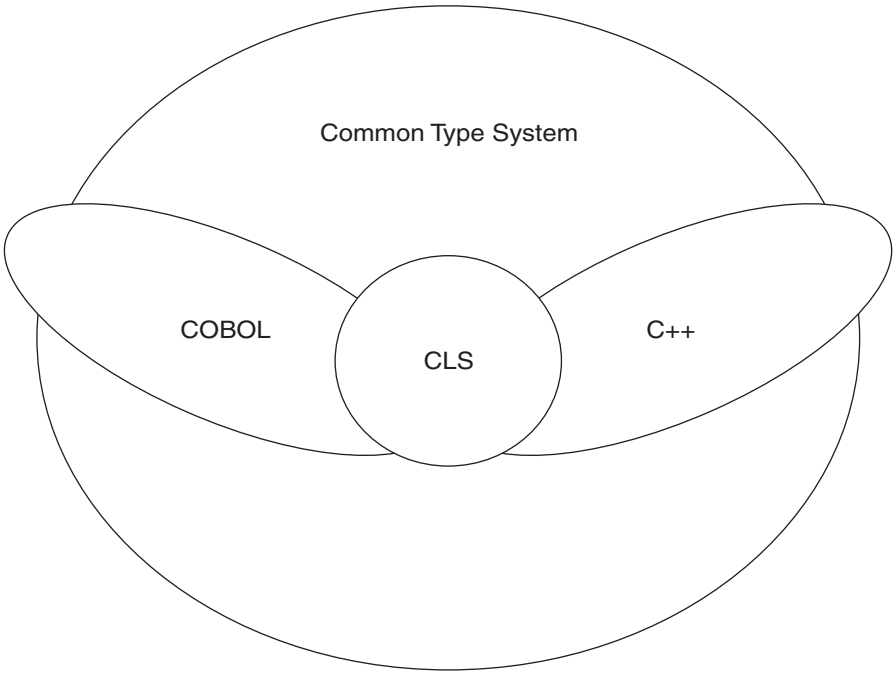

Figure 1-2 Relationship of Languages, the CTS, and the CLS

Fujitsu COBOL, and Microsoft Managed C++, use the CTS and CLS. One of the things it shows is that some aspects of both languages are not CTS compliant. It also shows that the Common Type System is too big for any single language.

The CLS is actually a set of restrictions on the CTS. The CLS defines not only the types allowed in external calls, but the rules for using them, depending on the goal of the user. There are three categories of those who might wish to comply with the CLS: frameworks, extenders, and consumers. **Frameworks** are libraries that are intended to be language- and system-independent. Extenders and consumers are both targeted at language developers. **Consumers** intend their languages only to use frameworks supplied by others. **Extenders** intend to extend their language—define types based on base types, define language-based libraries, etc.

Information on the CLS is contained in Partition I (Chapter 2 of this book), in the following sections:

- Section 7 is a discussion of frameworks, consumers, and extenders.
- Section 8 describes CLS rules in the context of the CTS discussion because the CLS is a subset, with restrictions, of the CTS.

Miller-CHP1.fm Page 6 Wednesday, September 24, 2003 1:50 PM

- Section 10 explains name and type rules for the CLS.
- Section 11 is a list of the CLS rules only.

Metadata

Along with the Common Type System, metadata is at the heart of the CLI. CLI-compliant compilers generate metadata and store it with the code in the executable file according to the file format specified in Partition II, sections 21–24 (Chapter 5 of this book). Metadata describes the code by describing the types that the code defines and the types that the code references externally. There is enough information stored in the metadata to (among other things):

- Manage code execution (load, execute, manage memory, and inspect execution state)
- Install code, resolve implementation versions, and perform other administrative functions
- Enable cross-language operation

Metadata provides information to tools like debuggers and verifiers, and it permits communication between tools. Among the information the metadata contains is the following:

- A description of the deployment unit (the assembly), called the **manifest**
- Descriptions of all of the types in each module
- Signatures of all methods in each module
- Custom attributes in each module

Figure 1-3 shows some of the possible users of metadata.

One of the services of the CLI that uses metadata is called **reflection**. In the CLI, a reflection object represents a type, describes it, and is an in-memory representation of the type—but is not an instance of the type. Reflection allows you to discover all of the information contained in metadata about a type, including the name of the type, its fields, its methods, whether it is public or private, any attributes attached to it, and more.

Metadata simplifies the job of developing designers, debuggers, profilers, type browsers, and other tools by making the information they need easily available without making them walk the code to find it. Metadata is the key that enables cross-language programming. When a compiler reads a module compiled in a different language, it reads the metadata, thus allowing seamless interoperation, much as a traditional compiler might read a "header file" describing the other module; but by reading the metadata directly from the module, there's no possibility of reading the wrong header file and there is no need to generate a separate header file to describe your own code.

Miller-CHP1.fm Page 7 Wednesday, September 24, 2003 1:50 PM

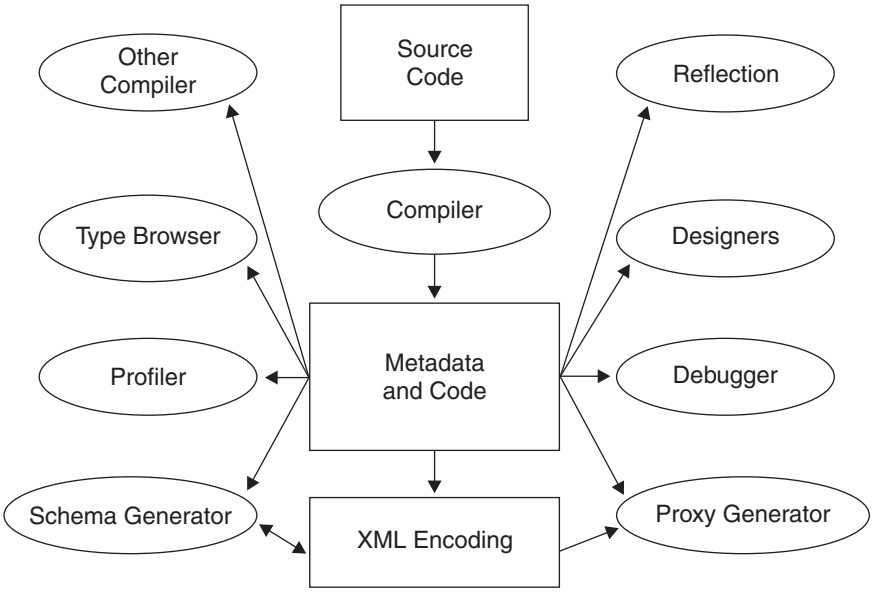

Figure 1-3 Users of Metadata

Custom attributes were designed as part of the CLI to allow extensibility without requiring languages to continue to add new keywords. Custom attributes include markers for CLS compliance, security, debugging, and many language- and tool-specific attributes. Some custom attributes are defined by the CLI, but they can also be defined by a compiler or by the tools that use the attribute. Languages that identify as distinct types what the VES sees simply as 32-bit integers—for example, C++, which distinguishes **int** from **long** even when both are 32-bit—would create a custom attribute that would identify the different types to the compiler.

Custom attributes are essential to tools. In a programming environment with designers, you might create a new button object. A custom attribute would tell the designer that this object is a button, and at runtime the designer would list it as one of the available buttons.

If the VES that you're using includes a proxy generator, you could make an object available externally to a Web service by putting in a custom attribute telling the proxy generator to create a proxy for the object, and another attribute telling the Web service that the proxy should be included. Metadata stores custom attributes, making them readily available to any tool.

Miller-CHP1.fm Page 8 Wednesday, September 24, 2003 1:50 PM

Section 9 of Partition I (Chapter 2 of this book) is a discussion of the architecture of metadata. Sections 1–20 of Partition II (Chapter 3 of this book) define an assembly language based on the CIL, using it to define the semantics of metadata.

Execution and Deployment Models

Assemblies are the unit of deployment for one or more modules. They are not a packaging mechanism and are not intended to be an "application," although an application will consist of one or more assemblies. An assembly is defined by a **manifest**, which is metadata that lists all of the files included and directly referenced in the assembly, what types are exported and imported by the assembly, versioning information, and security permissions that apply to the whole assembly.

Although the compilers capture the versioning and security information, it is the implementation of the VES that allows users to set policies that determine which versions are to be used, and how security is implemented. An assembly has a security boundary that grants the entire assembly some level of security permission; e.g., the entire assembly may write to a certain part of the disk, and methods can demand proof that everyone in the call chain has permission to perform a given operation.

The notion of "application" is not part of this standard. It is up to the implementer to determine how applications relate to assemblies. The standard does, however, encompass the idea of an **application domain**.

A process may have more than one application domain. Assemblies are loaded into application domains. Information about the available classes, the associated code, and the static variables is housed in that application domain.

The execution model is that compilers generate the Common Intermediate Language (CIL). How and when the CIL is compiled to machine code are not specified as part of the standard, and those determinations rest with the implementation of the VES. The most frequently used model is just-in-time (JIT) compilers that generate native code as it is needed. Install-time compilers are another option, and it is also possible to implement an interpreter, rather than a compiler, for the CIL. Native unmanaged code would go directly to the VES. Figure 1-4 illustrates the execution model.

Chapter 6 of this book contains Partition III, which includes a complete reference to CIL instructions.

Metadata and the File Format

The file format specified by the CLI is an extension of the standard PE (Portable Executable) file. The information required to write a PE file for managed code, and the logical and

Miller-CHP1.fm Page 9 Wednesday, September 24, 2003 1:50 PM

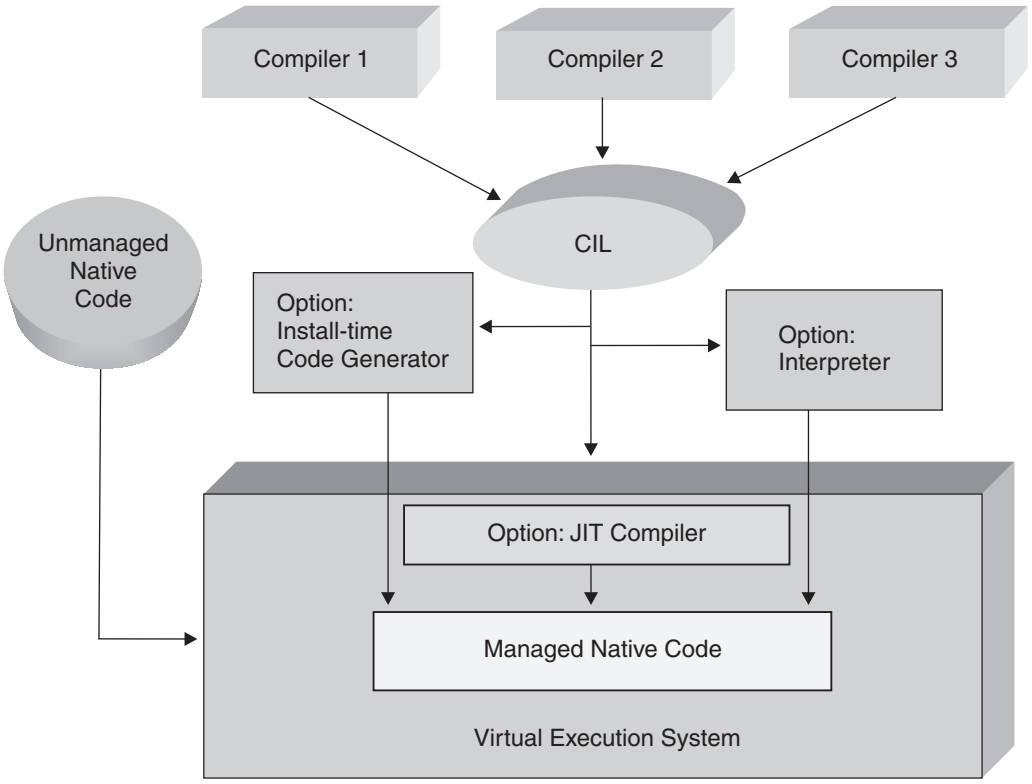

Figure 1-4 Execution Model

physical layout of metadata within a file, are described in detail in Chapter 5 of this book, which covers sections 21–24 of Partition II. Chapter 4 of this book provides a road map to the very dense, but important, information in those sections. Chapter 4 also contains a detailed diagram of the metadata tables and an annotated dump of a very small managed PE file, as well as a map of the metadata.

The Virtual Execution System

The Virtual Execution System (VES) provides an environment for executing managed code. It provides direct support for a set of built-in data types, and it defines a hypothetical machine with an associated machine model and state, a set of control flow constructs, and an exception handling model. To a large extent, the purpose of the VES is to provide the support required to execute the Common Intermediate Language instruction set, described in Partition III (Chapter 6 of this book).

Understanding the VES is important not only to those interested in implementing a VES, but to compiler designers. Compiler designers need to know what the VES does and does

Miller-CHP1.fm Page 10 Wednesday, September 24, 2003 1:50 PM

not do. This book clearly lays out those lines. Although there is information on the VES throughout the standard, section 12 of Partition I, and all of its subsections, contains important information on the VES, as does Partition III. The early sections of Partition IV (Chapter 7 of this book) are valuable for VES developers because they explain what must be implemented if a staged development is required.

For VES developers, there is a suggested order for reading the standard. If you understand the basic notion of a Virtual Execution System and managed code, read Partition III (Chapter 6) first, because it describes the instructions of the CIL, which are at the heart of the VES. Then go back and read Partition I, and try to fill in the architectural details. Finally, go on to the other sections.

The Standard Framework

The Standard Framework adheres to the CLS, and is intended to be cross-language. Both the libraries and the execution environment are factored: being able to use certain libraries requires corresponding pieces of the execution environment.

A **profile** is a set of libraries, grouped together to form a consistent whole that provides a fixed level of functionality. A basic CLI profile, the Kernel Profile, includes the C# class library. The Compact Profile includes those libraries and, in addition, libraries for XML and reflection, enough for scripting languages like ECMAScript (better known under the product names "JScript" and "JavaScript").

Optional libraries include floating point numbers, decimal arithmetic, and multi-dimensional arrays.

Each library specifies the available types and how they are grouped into assemblies, the set of CLI features that support that library, and any modifications in the library to types defined in other libraries, such as the addition of methods and interfaces to types defined in other libraries and additional exceptions that may be thrown by this library. A **framework**, a term used extensively in this standard, consists of simply the available type definitions, without the additional supporting information.

The Standard Framework is published in a companion to this book, entitled the *.NET Framework Standard Library Annotated Reference*.