MLLIF

Enabling Multiple Languages to Interoperate Within a Single Process

Yeong-won Seo

Abstract

Cross-language interoperability is a persistent challenge in modern software development, where applications often integrate languages like C++ and C# for game development or Python and CUDA for machine learning to harness their unique strengths. Existing tools, such as IKVM.net, which connects Java and .NET but is restricted to Java 8, and Jython, which runs Python on the Java Virtual Machine but remains stuck at Python 2.7, are outdated and insufficient. Modern runtimes like .NET Core also face difficulties replacing legacy systems, such as the .NET Framework and Mono used in the Unity Engine. Furthermore, traditional inter-process communication (IPC) frameworks like gRPC introduce significant overhead, making them impractical for performance-critical or low-latency applications, thus necessitating a more efficient approach.

This paper introduces an innovative method to automate cross-language interoperability using MLIR and LLVM infrastructure. Unlike conventional techniques, this solution automatically generates code for seamless function calls and data exchange across languages, reducing manual effort significantly. It eliminates the need for hand-coded bridges, ensures memory safety, and maintains consistent garbage collection. In a case study combining C++ and C#, it streamlined data sharing, automated bridge creation, and minimized complexity and errors.

This approach offers platform independence, high performance, and compatibility with native compilers and runtime environments, making it highly adaptable across diverse ecosystems. By providing a scalable, efficient way to integrate multiple languages, this method accelerates development cycles, enhances software robustness, and establishes a new benchmark for cross-language interoperability, addressing modern software development challenges effectively.

1. Introduce

The Unity Engine exemplifies the challenges of cross-language integration in modern software development, leveraging C/C++ for performance-critical components and C# for user-defined scripts and less-demanding tasks.

In C/C++, developers manually manage memory, whereas C# relies on a garbage collector (GC) to automate this process. Unity adapts the Mono Virtual Machine (VM) to execute C# assemblies, employing the Boehm GC, a conservative garbage collector that, unlike more advanced GCs, does not realign memory blocks to mitigate fragmentation [1] [2]. This design choice allows Unity to manage objects as raw pointers in the marshaling layer between C# and C/C++, simplifying data exchange but sacrificing flexibility.

The evolution of .NET has exposed the limitations of this approach. Following the announcement of .NET 5 in 2020, aimed at unifying fragmented development environments, the Mono project transitioned to CoreCLR, the runtime powering modern .NET versions. CoreCLR's GC, which realigns memory blocks for efficiency, conflicts with Unity's reliance on direct pointer manipulation, revealing technical debt from its lack of an abstraction layer. Since adopting .NET 4 in Unity 2017.1 [3], the engine has struggled to incorporate advancements beyond .NET Standard 2.1 [4], even as .NET 9 emerged by 2024. Similarly, other engines like CryEngine, also tied to the Mono-based .NET Framework [5], face difficulties modernizing legacy codebases that mix multiple languages.

These challenges underscore the broader problem of cross-language interoperability, where legacy systems hinder efficiency and adaptability. This paper proposes a novel solution to automate integration across language boundaries, addressing such limitations head-on.

2. Prior approaches

Approaches have been explored to enable interoperability between two or more programming languages as Table 1 (while inter-process communication, or IPC, is noted, its network overhead makes it less relevant here)

2-1. Hybrid Language Model

C++/CLI exemplifies the hybrid language model, developed by Microsoft as part of Visual C++. Initially, Microsoft introduced Managed Extensions for C++ in Visual C++ 2002 to bridge C++ and .NET. However, bridging different memory models made it challenging to incorporate the latest C++ features and standard library updates. Consequently, Microsoft deprecated Managed Extensions in 2004 and released C++/CLI in 2005. As of 2023, while the C++26 standard is in development and supported by compilers like Clang and GCC, C++/CLI remains limited to C++20 (finalized in 2020), restricting its relevance as C++ evolves.

Another example, Cython, emerged in 2007 to connect C/C++ and Python. Though actively maintained,

Approach	Example
Implementation of a new language with a hybrid nature of two languages	C++/CLI, Cython
Communication between processes (IPC)	Unix Socket, Pipe
Reimplementation on other VM	Jython, IronPython
Interoperability using FFI	P/Invoke, JNI

Table 1: Prior Approaches Summary

Cython implements its own parser and lexer, requiring users to learn a distinct syntax separate from C/C++ and Python [6]. This additional learning curve reduces its accessibility. Moreover, such hybrid frameworks are typically available only for widely adopted languages like C++ and Python, limiting their applicability.

2-2. Reimplementation on other VM

Reimplementing a language on another virtual machine poses significant challenges. For example, Jython runs Python on the JVM but supports only Python 2.7 (released in 2010) [7], while IronPython, targeting the CLR, is limited to Python 3.4 (released in 2014) [8]. Despite ongoing updates, both projects lag nearly a decade behind current Python standards, reflecting their slow development pace. Similarly, IKVM.NET enables Java bytecode to run on the CLR by implementing the JVM and Java Class Library within .NET, yet it remains stuck at Java 8 (released in 2014) [9]. These delays stem from the substantial resources required to maintain compatibility with evolving language standards.

2-3. Interoperability using FFI (foreign function interface)

The Foreign Function Interface (FFI) provided directly by a language enables it to call functions from external languages. Unity Engine utilizes FFI to integrate C# with C++ [1]. However, a key challenge is that function signatures (such as symbol names) must be manually defined, and type marshaling must be explicitly managed.

While Unity Engine addressed the first issue by automating function signature definitions with scripts, it did not resolve the latter. More importantly, while FFI can bridge native languages with managed languages or even connect native languages with each other, it struggles to handle interoperability between managed languages. This is because FFI primarily supports dynamic libraries that can be loaded by the operating system.

Existing approaches to cross-language interoperability—hybrid language models, VM reimplementations, and FFI—each face significant limitations. Hybrid models demand new syntax and are confined to specific language pairs, VM reimplementations lag in adopting new standards due to resource demands, and FFI requires manual effort while struggling with managed-to-managed language integration. Even IPC, though useful in some cases, introduces performance overhead. These shortcomings highlight the need for a more efficient, automated, and flexible solution.

To address these challenges, this paper proposes an approach that is:

- Platform-independent
- Dependent only on the languages' native FFI
- Single-process
- Automated in bridge generation

Central to this approach is MLLIF (MLIR-based Language-to-Language Interoperability Flyover), a framework leveraging MLIR and LLVM to enable seamless cross-language integration. The full source code is available at https://github.com/mllif/mllif-project .

3. Main Body

3-1. Structure

MLLIF enables interoperability by integrating into the compilation and invocation processes of two languages: the callee (the language being called) and the caller (the language making the call). The callee language must first be compiled into an assembly—defined here as the executable output, such as a DLL from C++ source code—before it can be invoked.

During compilation, MLLIF intercepts the callee language's process to extract its ABI (Application Binary Interface) and API (Application Programming Interface) details, collectively termed the Symbol Model. This step produces two outputs: the assembly and the **Symbol Model**. From the Symbol Model, MLLIF generates an interface, called the **Bridge**, implemented using the caller language's Foreign Function Interface (FFI). The Bridge enables the caller to invoke the callee seamlessly.

To ensure compatibility, the callee's calling convention is wrapped in a standardized format, typically the C calling convention, creating what is termed the **Wrapper**. The Wrapper depends on the callee's runtime—defined here as the execution environment, including virtual machines like the JVM or CLR for managed languages (e.g., Java or C#) or the native execution state for unmanaged languages (e.g., C/C++). For instance, wrapping C# in a C calling convention involves a shared library exposing C# interfaces via CLR Native Hosting. However, for languages like C/C++, which natively use a general calling convention, the assembly itself serves as the Wrapper, bypassing additional runtime dependencies.

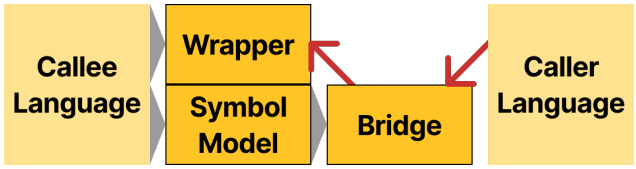
This process splits into two phases:

- **Callee-dependent**: Generates the Symbol Model and Wrapper.
- **Caller-dependent**: Generates the Bridge.

These phases are handled by distinct components: the **Frontend** produces the Symbol Model and Wrapper, while the **Backend** generates the Bridge. Invocation follows three steps:

1. The caller language invokes the Bridge.
2. The Bridge invokes the Wrapper.
3. The Wrapper invokes the callee's assembly.

This structure, illustrated in Figure X (showing the call path and compilation workflow), ensures that any language supporting FFI can call another, regardless of their runtime differences, provided both support FFI.



- [Figure X] Call-path and rough compile-path

(Note that Red arrow means call-path, Gray or blue arrow means compile-path)

3-2. Interoperability between native language and managed language

Section 3-1 demonstrated that MLLIF enables any FFI-supporting language to call another. However, a key goal of this paper is to reduce the development costs—specifically, the manual effort required to implement and maintain cross-language bridges. This section details MLLIF's interoperability between native languages (e.g., C/C++) and managed languages (e.g., C#), showing how automation minimizes these costs.

3-2-1. LLVM and MLIR LLVM (Low-Level Virtual Machine) is a compiler infrastructure that uses an intermediate representation (IR) to generate platform-independent executable code from various languages [10]. MLIR (Multi-Level Intermediate Representation), an extension of LLVM IR, integrates multiple IR levels—from high-level abstractions to low-level operations—supporting diverse computation domains [11]. In MLLIF, MLIR's flexibility enables the extraction and transformation of language-specific details into a unified form, while LLVM compiles this into executable assemblies.

For native languages, modern compilers like flang (FORTRAN) and clang (C/C++) leverage MLIR dialects—FIR for FORTRAN and ClangIR for C/C++—which are then lowered to LLVM IR and compiled into assemblies. MLLIF builds on this infrastructure to automate interoperability, avoiding the need for separate compilers per language.

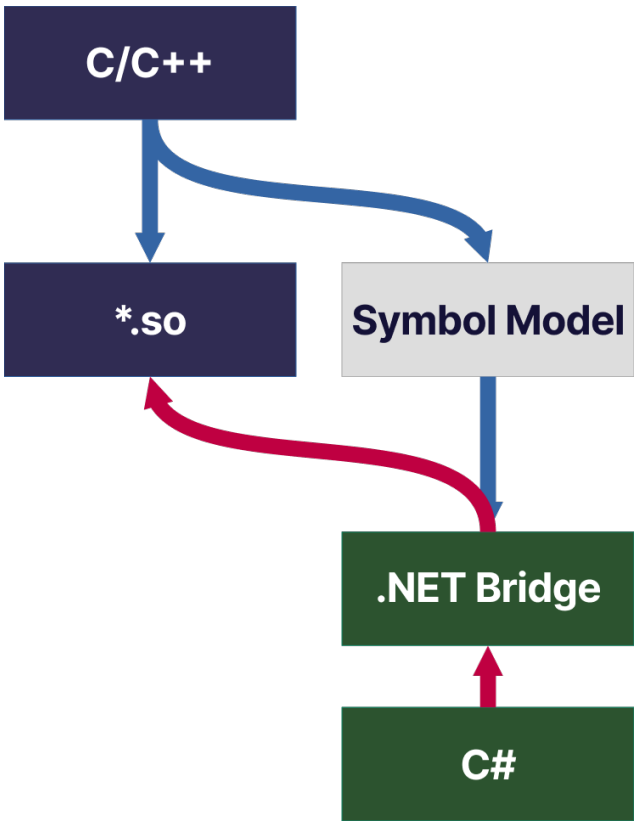
3-2-2. C/C++ Frontend In MLLIF, the C/C++ frontend automates Symbol Model generation for native languages. Using clang, C/C++ source code is compiled into MLIR (ClangIR dialect). However, this process typically discards details like function forms (e.g., member vs. global), names, and parameters. To retain this, MLLIF integrates as a clang plugin, annotating Abstract Syntax Tree (AST) nodes with standardized Annotate Attributes during compilation.

The resulting MLIR is processed by the MLLIF-MLIR module (a shared component across native languages), which extracts critical details: object sizes and alignments, parameter names and types, return types, and function metadata. These are stored in the Symbol Model, encoded as XML to preserve hierarchical structures like classes and namespaces. The MLIR is then lowered to LLVM IR via mlir-opt and compiled into a dynamic library (assembly) using LLVM, all automated to eliminate manual ABI extraction.

3-2-3. C# (.NET) Backend The C# backend generates the Bridge for managed languages, using P/Invoke as C#'s FFI to call C/C++ assemblies. From the Symbol Model's XML, MLLIF auto-generates C# code:

- **Data Types**: Applies StructLayoutAttribute to structs, matching C/C++ memory layouts (noting C# distinguishes structs from classes, unlike C/C++).
- **Functions**: Uses DllImportAttribute to specify the dynamic library, calling convention, and symbol name, enabling the .NET runtime to load and invoke the assembly.

This recursive process converts the entire Symbol Model into a C# Bridge, compiled into a .NET assembly compatible with any .NET version or language (e.g., VB.NET). By automating Bridge creation, MLLIF eliminates manual P/Invoke coding, reducing development effort. Figure Y illustrates this C# to C/C++ call path, showing seamless integration across runtimes.



- [Figure Y] C# to C/C++ call-path

3-2-4. Vice Versa : Call C# from C++ Section 3-2-3 demonstrated C# invoking C++ via MLLIF's automated Bridge generation. This section addresses the reverse—enabling C++ to call C#—using .NET's Native Host functionality, while reducing development costs through automation. MLLIF leverages Roslyn Source Generators and existing MLLIF-MLIR components to eliminate manual wrapper creation.

C++ loads the .NET runtime via Native Host, a stable API that supports any .NET version via external configuration, avoiding compile-time version locks. However, since C++ cannot directly call .NET member functions, MLLIF uses Roslyn's Source Generator feature to wrap these as static proxy functions during C# compilation.

3-2-4-1. Data Marshaling and GCHandle C# unmanaged types (e.g., `int`, `float`, `struct` with unmanaged fields) are natively compatible with C/C++ without special marshaling. For managed reference types (e.g., classes), MLLIF employs `GCHandle` (`System.Runtime.InteropServices.GCHandle`), which pins .NET objects in memory and provides a stable pointer via `GCHandle.ToIntPtr`. This pointer, convertible back with `GCHandle.FromIntPtr`, allows C++ to reference and invoke methods on .NET objects. MLLIF wraps these objects as structs containing a GCHandle field, with `System.Object` acting as a minimal interface (akin to COM's `IUnknown`) on the native side.

3-2-4-2. Arrays and Strings via Pinning

Note that this section (3-2-4-2) covers optimisation techniques specific to a particular framework (.NET). In other words, it is optional and may not always apply to all backends or frontends of MLLIF.

For arrays and `System.String`, MLLIF uses `GCHandle` with pinning to prevent CoreCLR's Garbage Collector from relocating memory. `GCHandle.AddrOfPinnedObject` provides direct access to the buffer address, treating `System.String` as an unmanaged `char*` and array elements as bridge types. MLLIF automatically pins these objects during marshaling, ensuring efficient and safe access from C++ without manual memory management.

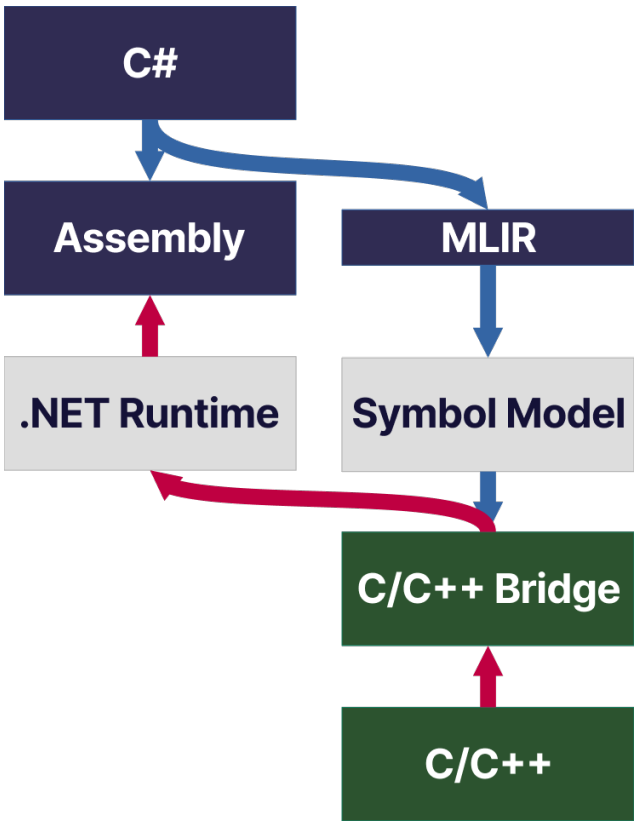
3-2-4-3. Generation of Wrapper MLLIF integrates into Roslyn to generate C# proxy functions and types, producing a shared library (Wrapper) callable from C++. The Source Generator creates C++-compatible code that invokes .NET assemblies via the runtime. This code is processed by the C/C++ frontend (Section 3-2-2), reusing MLLIF-MLIR to generate both the Symbol Model and Wrapper assembly, automating what would otherwise be labor-intensive.

3-2-4-4. C++ Backend The C++ backend generates header files from the Symbol Model, including `extern` function declarations and type definitions matching C++'s basic types. This recursive process mirrors the C# backend (3-2-3), requiring no additional conversions, enabling seamless use in C++.

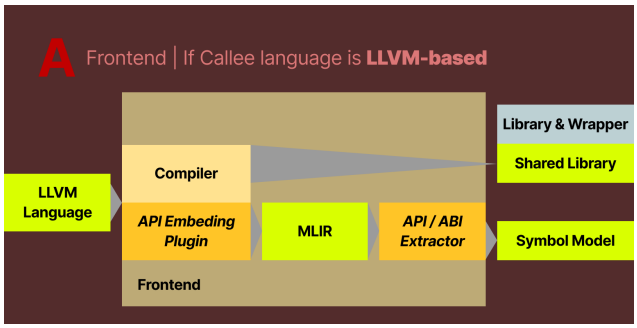
As shown in Figure Z:

1. C/C++ calls the Wrapper assembly
2. The Wrapper invokes the .NET runtime
3. The runtime the proxy function
4. The proxy function invokes the target C# function

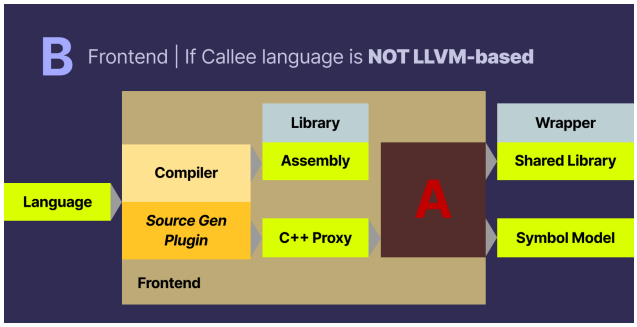
This four-step process, fully automated by MLLIF, enables C/C++ to call C# efficiently, reducing development overhead.



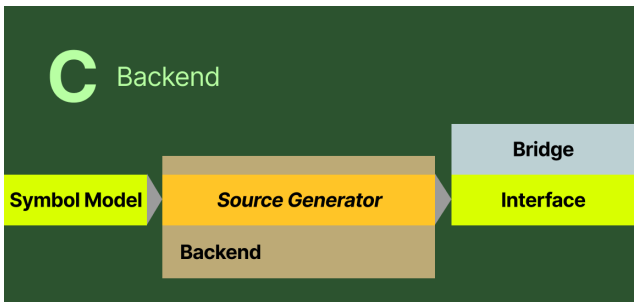
- [Figure Z] C/C++ to C# call-path



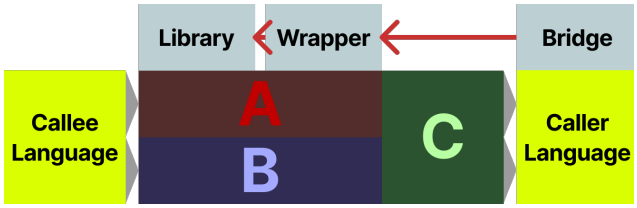
- [Figure W-a] Full diagram of frontend for LLVM languages



- [Figure W-b] Full diagram of frontend for non-LLVM languages



- [Figure W-c] Full diagram of backend



- [Figure W-d] Full compile/call path

4. Conclusion

This paper proposes MLLIF (MLIR-based Language-to-Language Interoperability Flyover), a framework that automates cross-language compatibility while delivering high performance in a single - process environment. Addressing the limitations of traditional approaches—such as manual bridge coding in hybrid models, slow updates in VM reimplementations, and FFI's inefficiencies (Section 2) — MLLIF offers the following contributions:

- Platform-independent compatibility: Leveraging MLIR and LLVM, MLLIF creates a standardized Symbol Model that abstracts ABI and API details for both native (e.g., C/C++) and managed (e.g., C#) languages, ensuring seamless operation across platforms.
- Automated bridge generation: By processing the Symbol Model through language-specific backends, MLLIF eliminates manual coding, reducing development costs as demonstrated with C# and C/C++ (Section 3-2).
- Scalability: The reusable Symbol Model and Bridge structure supports expansion to additional languages, such as Python or Java, with minimal adaptation.
- Performance optimization: Single-process execution avoids network overhead, enhancing efficiency over alternatives like gRPC.

MLLIF's flexibility shines in bridging native and managed languages, as shown with Unity-like scenarios (Section 1), and extends to managed-to-managed interoperability, broadening its applicability.

4-1. Expected Effect

MLLIF overcomes technical barriers to cross-language integration, streamlining software development by automating interoperability tasks. In practice, it could

halve bridge creation time — easing Unity's .NET transition woes — and boost efficiency across diverse projects. This research lays a foundation for future advancements, potentially standardizing language interaction in domains like gaming and machine learning.

References

[1] J. PETERSON, "Porting unity to CoreCLR," 2023. [Online]. Available: https://web.archive.org/web/20240723152903/https://unity.com/blog/engine-platform/porting-unity-to-coreclr. [Accessed: 23-Jul-2024].

[2] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Softw. Pract. Exper.*, vol. 18, no. 9, pp. 807–820, Sep. 1988.

[3] "Using .NET 4.x in unity," 2022. [Online]. Available: https://web.archive.org/web/20221102202915/https://learn.microsoft.com/en-us/visualstudio/gamedev/unity/unity-scripting-upgrade. [Accessed: 02-Nov-2022].

[4] ".NET profile support," 2025. [Online]. Available: https://web.archive.org/web/20250314124121/https://docs.unity3d.com/6000.0/Documentation/Manual/dotnet-profile-support.html. [Accessed: 14-Mar-2025].

[5] "CryMonoBridge." [Online]. Available: https://web.archive.org/web/20250221073945/https://www.cryengine.com/docs/static/engines/cryengine-5/categories/23756813/pages/26874559. [Accessed: 21-Feb-2025].

[6] "Cython language basics," 2025. [Online]. Available: https://web.archive.org/web/20250209181052/https://cython.readthedocs.io/en/latest/src/userguide/language_basics.html. [Accessed: 09-Feb-2025].

[7] "Jython." [Online]. Available: https://web.archive.org/web/20250124083053/https://github.com/jython/jython. [Accessed: 24-Jan-2025].

[8] "IronPython." [Online]. Available: https://web.archive.org/web/20250301031043/https://github.com/IronLanguages/ironpython3/. [Accessed: 01-Mar-2025].

[9] "IKVM." [Online]. Available: https://web.archive.org/web/20250126181605/https://github.com/ikvmnet/ikvm. [Accessed: 26-Jan-2025].

[10] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *International symposium on code generation and optimization, 2004. CGO 2004.*, 2004, pp. 75–86.

[11] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, "MLIR: Scaling compiler infrastructure for domain specific computation," in *2021 IEEE/ACM international symposium on code generation and optimization (CGO)*, 2021, pp. 2–14.