Programming Language Interoperability

Todd Malone
Division of Science and Mathematics
University of Minnesota, Morris
Morris, Minnesota, USA 56267
malon153@morris.umn.edu

ABSTRACT

A discussion of systems and methods for the interop of programming languages

Keywords

interoperability, language interoperability, programming languages, virtual machines, markup languages

1. INTRODUCTION

Interoperability, colloquially shortened to interop, is the ability for two or more systems to work together. This definition is very broad, covering anything from groups of people to businesses or bureaucratic systems to pieces of hardware. This paper will discuss interoperability between programming languages. While this narrows the field significantly, programming language interop is still a very broad topic.

For instance, all browsers perform some kind of interop. Information to be displayed on a web page is handled by HTML, and while formatting computation handled by PHP or JavaScript; if there is more computation to be done, or if a database needs to be consulted, these front-end languages will communicate with a server, often written in languages like Java or C.

Likewise, these servers may be written in several languages themselves. A blog hosting service, Kidblog[1], uses a backend written in both Java and Clojure, using Java to handle translation between the front and back ends, and Clojure to handle the actual server load.

The challenges that arise when dealing with interoperability can be many, and sometimes surprising[3]. The tools and approaches used to overcome these challenges are likewise many. In the interest of brevity, we will look at a small subset of these challenges, along with several tools used to deal with them.

In Section 2 I explore further why interoperability is desirable, what advantages it can confer, and where it can be useful. This will touch on differing language capabilities, ease of use, hardware independence, and implications for distributed computing.

Next, in Sections 3 and 4, I describe two particular tools that are commonly used to achieve interoperability. These tools will be virtual machines, paying particular attention to the Java Virtual Maching (JVM) and .NET's Common Language Runtime (CLR), and markup languages, with focus on two particular systems that make use of markup languages, Starlink and FML.

Section 5 will detail particular challenges involved with interoperability and approaches to handling them. Following a general description of the approach, I will describe how each tool can be used to implement that approach, along with strengths and weaknesses of each setting.

Section 6 will look at a few of the performance implications of the tools discussed.

[1]kidblog.org

UMM CSci Senior Seminar Conference, April 2014 Morris, MN.

2. INTEROPERABILITY

The primary importance of interoperability lies in the varying capabilities of different languages. Programming languages perform best in particular circumstances, and different languages perform better in different situations.

Some languages are constructed to tackled specific problems, but provide almost no support for others. Erlang, for instance, is one of the primary languages used in distributed computing. It was built around remote procedure calls (RPCs), and handles them simply and powerfully. However, manipulating strings in Erlang is very difficult, because Erlang was never intended to deal with strings. As a result, it has no library for string interaction, and treats them as arrays of characters. A system using Erlang for handling distributed communications which also needed to manipulate strings would have use for interop: another language could be used to handle the strings, while leaving the RPCs to Erlang.

Still others allow access in ways that are hidden in other languages. The memory access model in C, for instance, differs from many other high-level languages presently in use. C enables very fine-grained memory access and control, and places all of this control in the programmer's hands. Likewise, assembly languages allow direct access to hardware and the underlying storage mechanisms. The higher level of control offered allows developers to make very specific, fine-tuned adjustments to their systems, enabling performance improvements in ways specific to the program or the hardware platform. However, these languages have high learning curves, and programs using them may have security concerns

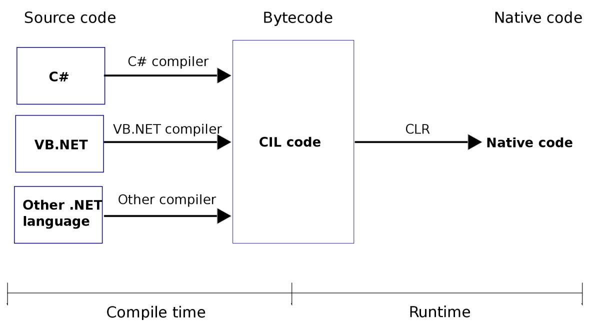

Figure 1: VM architecture in the CLR[8]

if not done correctly.

The difficulty involved in learning a language is not a minor concern, either. Developer time and energy are important considerations when designing a system. For example, Java was initially intended to be the go-to language for web development. In practice, most web development now is done with scripting languages like Ruby or JavaScript[2]. Java has, in turn, become more of a server back end language. The reasons for this shift had nothing to do with what Java was capable of, and much more to do with what programmers were willing to deal with. The learning curve for Java is higher than many scripting languages, and is generally much more verbose.

Apart from the capabilities of the languages involved, interoperability is often tied to hardware independence. The problems faced in achieving language independence and hardware independence are frequently similar, or have similar solutions. As noted above in the case of Java, designing for hardware independence allowed the JVM to act as a tool for language independence as well.

In client/server architecture, such as in Kidblog, the client-side and server-side environments are likely written in different languages, and guaranteed to be on different machines. Kidblog handles communication between the two using JSON, a markup language. On the client-side, JavaScript marshals blog posts into JSON objects, which are then sent to the server. There, they are converted to Java objects, which Clojure can then act on. Since Java is more strongly typed than JavaScript, JSON is also used to help translate between the two. We'll take a closer look at this in Section 5.1.

3. VIRTUAL MACHINES

The term virtual machine (VM) refers to software that models the operating system of a computer or the underlying physical hardware. Effectively, VMs are indistinguishable from the OS or hardware they are emulating, which allows them to run programs not runable by the native OS, or with restrictions not enforced by the native OS or hardware. The focus of this paper will be on virtual machines that act as a runtime environment for a single process, sometimes called process VMs. These VMs in particular enforce restrictions on what system resources their hosted process has access to. Specifically, they typically provide restricted, VM managed memory spaces, ensuring that hosted processes can't access memory beyond what is allocated to it.[12] is this relevant?

[2]Unrelated to Java

There are two major advantages to virtual machines. The first of these is a type safety mechanism. Virtual machines such as the Java Virtual Machine (JVM) and .NET's Common Language Runtime (CLR) have a base set of types (in the CLR, this is actually called the Common Type System, or CTS. It is implicit and unnamed in the JVM). The VMs can use these type systems to ensure the correctness of participating languages and gives languages a common ground for their type information.

The second advantage lies in the VM's intermediate language. These are low-level languages, designed to be both compact and efficiently compiled to machine code, and are what is actually run by the virtual machine. As shown in Figure 1, any high-level language can be run on a virtual machine, as long as there is a compiler to translate that language into the VM's intermediate language.

These languages are still an abstraction of machine code, and so they still require an interpreter or a compiler for execution. The VMs also contain interpreters for their intermediate language, but also utilize a run-time optimization method called Just-In-Time (JIT) compilation. Most of the code is still interpreted, but portions of code are monitored for frequent use. High-usage sections of code are compiled to machine language for quicker access. This allows for specific optimizations based on the current input and hardware, which are rarely known before execution. [11]

The JVM and CLR will be our main subjects when discussing virtual machines. While similar in certain ways, these two VMs were designed for different purposes.

The JVM[3] was designed alongside Java for the purpose of hardware independence. Since the intent of Java was to be run off the web, there was no way of knowing what kind of hardware would end up running a piece of code, or even what kind of browser it would be running from. They needed to be able to compile to machine code only when the code was actually run, such that any hardware would be able to compile and run the code correctly. The solution was the JVM. The eventual discovery that other languages could make use of Java's intermediate language and the JVM was completely accidental.

In contrast, the CLR was developed with a host of languages, and was designed in part with the intent of facilitating interoperability between them. The purpose of the virtual machine in this context was to intentionally provide the common ground for languages based on the CLR, giving them access to features promoting interop, such as the Common Type System and metadata engine (discussed further in Section 5.1).

4. MARKUP LANGUAGES

Markup languages are primarily used for describing data. There are a range of markup languages covering several use cases, from document display or creation to data transfer and storage. For instance, HTML (HyperText Markup Language) is used to convey content information for web pages, while TEXis a language used to describe the layout and formatting of text documents. Others, such as JSON and

[3]References to "The JVM" refer to the general concept of Java's virtual machine. There are many implementations of JVMs, which follow certain specifications but implement certain concepts in different ways and may support different operating systems. [9, 10]

```
1  <Types>
2    <Method>String<Method>
3    <URI>String<URI>
4    <Version>String<Version>
5    <ST>String<ST>
6    <MX>Integer<MX>
7    ...
8  </Types>
9
10 <Header type=SSDP>
11   <Method>32</Method>
12   <URI>32</URI>
13   <Version>13,10</Version>
14   <Fields>13,10:58</Fields>
15 </Header>
16
17 <Message type=SSDP_M−Search>
18   <Rule>Method=M-SEARCH</Rule>
19 </Message>
20
21 <Message type=SSDP_Resp>
22   <Rule>Method=HTTP/1.1</Rule>
23 </Message>
```

Figure 2: A Starlink MDL specification[2]

YAML, were designed for modeling data in ways consistent with object oriented design[4].

One of the major players in the world of markup languages is XML, the Extensible Markup Language. XML's method of describing data is to enclose the relevant data with a tag. Tags are easily distinguishable from the data they contain, and are considered separate from the data when the XML is read. Notice in Figure 2 that the fields (`Method, URI, Version`) are made distinct from the actual data with angled brackets. Also note that tags can be nested: lines 11 through 14 are contained within the `Header` tag. This allows XML to describe full records or objects, which will be important in Section 5.1.

The eponymous extensibility of XML comes in the form of defining new tags. Systems using XML are not constrained to using the tags built in to XML, but can create new ones based on the data they will be handling. This allows for the creation of new markup languages based on XML, but built for a specific purpose. HTML is one example of such a language. Two others, which we'll take a closer look at, are FML and Starlink.

Starlink [2] is actually more than just a markup language. It is a software framework designed to achieve distributed interoperability between existing systems with different communications protocols. One of the core features of Starlink is XML-based markup languages called the Message Description Languages (MDLs). MDLs are used to model incoming messages from a protocol as abstract messages: messages separated into their constituent parts and types. A translation logic specification, also written in XML, defines what parts in two MDLs can be translated to one another. Once Starlink has performed appropriate transformations based on the message type received, it can then marshal a message for the target protocol using the newly created abstract message.

The Fuzzy Markup Language[1] was not designed for interoperating systems at all. Fuzzy controllers handle converting crisp, or discrete, data into sets that cannot be defined as strictly. Fuzzy controlers may be used in things like automobile transmission control, where but was designed for both hardware and language independence. In effect, this makes it possible to use it in an interoperating system, if one had a distributed system where fuzzy logic would be a useful tool.

[4]JSON and YAML in particular are sometimes called Data Description Languages for this reason

5. DIFFICULTIES AND APPROACHES

When building a system with interoperating languages, there are several aspects of the system that designers need to account for. Of particular importance is the lowest common denominator (LCD) constraint.

The LCD in this case is the largest subset of concepts that can be translated across the languages in use. These concepts include data types and underlying data structures. For example, if you have a system wherein you want to use a key/value map, but one of the languages only has support for arrays. This restriction is part of the system's LCD, and limits the functionality of the overall system. Here, this can be dealt with in several ways: Remove the restricting language from the system, remove the use of maps and just use arrays instead, or find ways to model maps as arrays. The first two options are rarely preferable. If a particular language is present in a system, it usually has a contribution not easily matched by any other language. Likewise, maps may be the ideal way to store the particular data being used, while arrays will be insufficient.

Finding a method of translating a concept from one language to the other is preferable, but may not always be possible, or as straightforward as in the above example. In some cases, achieving the translation can mean a loss of information or a loss of precision. C, for instance, has no concept of booleans. Zero is treated as false, while any other number is treated as true. When interoperating with a language that does use booleans, precision of booleans is lost. A number could be just a number, or it could be True. On C's end, information could be lost if a true/false test is made to determine if a variable contains a number, and act upon that number if it does. If C recives a True from another language, it may read it as one. needs refinement.

For issues like these, there are two concepts that help enable this translation with minimal loss: metadata and standard interfaces. I'm not sure if I should refer to standards as interfaces, or how to phrase that

5.1 Metadata and Data Type Conversion

When passing data between two languages, a system must have a way of ensuring that the type systems of its component languages are respected. Additionally, it must ensure that type information is not lost when data moves from a strongly typed language to a weakly typed language, so that information is available when the data moves the other way.

Metadata is the method by which this is accomplished[5]. Simply put, metadata is data describing data. I'm not sure how to talk about this, or if this is plenty.

One of the most basic things an interoperating system needs to take into account are the data types involved. When passing data between two languages, a system needs to have a way of ensuring that the type restrictions of the LCD are respected. Ide and Pustejovsky [5] suggest metadata as the method to accomplish this, and in practice is what both styles use. Metadata is simply data about the data. This can be used to preserve type information when data is passed to a language that has no concept of the data types involved, or establish type information when passing data to a language that requires more strictly defined typing.

VMs and metadata

There are two primary ways virtual machines handle metadata: language type specifications and metadata files. Both of these are handled by a compiler at compile time. Once the system has been compiled to the intermediate language, there should be no syntactic difference between the different parts, as they are now all the same language. don't think this is true..

Strongly typed languages like Java or C# include explicit type information with all data. This is essentially a form of metadata, though it is mainly used by the language compilers when transforming code to the intermediate language. In weakly or dynamically typed languages like Ruby and Groovy, this metadata does not exist. Since these languages perform run-time type checks instead of at compilation time, significantly more code is generated by their compilers, and they cannot take advantage of the intermediate language's primitive type system (everything must be assumed to be an Object-type object)[5]. I need to confirm this is true for Groovy

Compilers for both the JVM and CLR also generate metadata along side the intermediate code. The JVM stores this metadata in the same file as the code, while the CLR stores it in a separate file in the same location. This metadata is used by the run-time interpreter for type checking and to assist with JIT optimization.

MLs and metadata

Markup languages are built with the concept of metadata in mind. A common feature of markup languages is the ability to contain or prepend data with tags, which can effectively act as metadata when used to transfer data between languages.

This is the area where markup languages really shine. Languages like XML can be configured to describe many custom types, and can nest these types to describe components of larger data structures, including descriptions of full objects.

I'm concerned much of the above may need to be transferred to 4. I should have enough here to fill the section if I talk about Starlink and FML, but this is something to keep in mind. When talking about Starlink, consider including at least the <Types> section of their figure 7

5.2 Standards and Interfaces

Metadata is the core of successful interoperability. But if two systems attempting to communicate are expecting differently tagged data, they will still fail to interoperate. Metadata alone is not enough.

One example of this, as demonstrated by Shetty and Vadivel [7], occurs when trying to process web page output from Java and .NET services. I'm not sure if this is an example I should actually use. Or if I have any examples here

VMs and Standardization

In a virtual machine system, the standard is frequently already set.[6] Virtual machines come with their intermediate language with its own libraries and type systems. These essentially form the standard: each language hoping to run on a particular virtual machine must be translatable to the intermediate language, which can be used as a sort of lingua franca. Any language built to run on the VM should be able, with little additional effort, be able to interoperate with any other language on that VM.

[5]While this does have performance implications[6], they do not outweigh the gains from utilizing the VM

[6]This is not always the case, but is common practice with recent VMs built with the goal of interop.

MLs and Standardization

Standards are not inherent to markup languages, and enforcement cannot be left to underlying hardware or compiler, as they can in a virtual machine setting. Some markup languages have do have standards built for them, such as schema for XML. These provide the same functionality as standards in virtual machines, but must be explicitly enforced by system designers, usually through third-party tools or manual additions to the system.

describe schema in detail with XML. Can maybe use Starlink as a working example, but just talk about schema for a while

Starlink

5.3 Error Handling

VMs and Errors

MLs and errors

6. PERFORMANCE

Regardless of how the system is built or the interop implemented, an interoperating system will always accrue some overhead. The cost of translating from one language to another can have surprising impacts on efficiency.

The main issue faced in virtual machines is the LCD constraint. Because all languages eventually end up running in the same language, care must be taken that the intermediate language imposes few constraints on the higher level languages.

Li, White, and Singer show that in the Java Virtual Machine, non-Java languages rely heavily on existing Java code libraries in order to mitigate performance difficulties. Additionally, they found that non-Java languages produced distinctly non-Java sequences of byte-code. While not tested, they indicate that JIT compilation optimization can potentially miss these sequences, as it is tuned to compile byte-code produced by Java.[6]

The CLR has no similar study, but because it was built to handle multiple languages at once, it is likely able to handle a wider variety of byte-code grammar efficiently. Should the previous paragraphs be one?

The primary concern for markup languages is in translation time. Because systems involving MLs usually also involve different languages at runtime, they also require translating between two or three languages during execution.

Bromberg et al's report on the Starlink framework, which handles three translations per message (two between standard communication protocols and Starlink's internal representation of them, one between internal representations of the two protocols in question), showed a non-negligible time lapse between message and response. The lowest of these was 255 milliseconds, but in a system passing many messages this can add up.

Ultimately, the performance costs of achieving interoperability must be weighed against the potential performance gains. In systems involving several specialized domains, or in systems utilizing diverse hardware, the gains can well outweigh the costs.

7. CONCLUSIONS

The two tool sets explored here have widely different applicable systems.

Virtual machines are much more feasible for systems being built from scratch, where all language decisions are in the hands of the developers. They may also be available to existing programs on a VM which a developer wishes to extend to a larger system; in this case, the extended system merely need be built on the same virtual machine, and it will be able to interoperate with the pre-existing software. More specifically however, virtual machines are most suited to systems that can exist on the same physical machine.

In comparison, markup languages are better suited to dealing with preexisting or legacy systems, where there is too much code to change or parts of the source are simply unavailable Does that happen?, and so rewriting is not an option. Likewise, if the existing system cannot target a particular virtual machine, perhaps because a compiler from that language to that VM doesn't exist, recompiling the existing program is not feasible.

Additionally, markup languages have an advantage in distributed system environments, where they can be used in sending data over the network. Network nodes have no reason or way to know what language other nodes are running, nor what hardware they are running on. A markup language can act as an intermediary in these cases, describing data in a language-free way.

These two cases are not mutually exclusive, and indeed, markup languages fill a gap that virtual machines simply cannot cover. A real-world system is more likely make use of both tools to cover different aspects of their system than to rely on one, as demonstrated by Kidblog.

8. REFERENCES

[1] G. Acampora. Fuzzy markup language: A xml based language for enabling full interoperability in fuzzy systems design. In G. Acampora, V. Loia, C.-S. Lee, and M.-H. Wang, editors, *On the Power of Fuzzy Markup Language*, volume 296 of *Studies in Fuzziness and Soft Computing*, pages 17–31. Springer Berlin Heidelberg, 2013.

[2] Y. Bromberg, P. Grace, and L. Réveillère. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 446–455, 2011.

[3] D. Chisnall. The challenge of cross-language interoperability. *Queue*, 11(10):20:20–20:28, Oct. 2013.

[4] J. Hamilton. Language integration in the Common Language Runtime. *SIGPLAN Not.*, 38(2):19–28, Feb. 2003.

[5] N. Ide and J. Pustejovsky. What does interoperability mean, anyway? Toward an operational definition of interoperability for language technology. In *Proc. 2nd Int. Conf. Global Interoperability Lang. Res*, 2010.

[6] W. H. Li, D. R. White, and J. Singer. Jvm-hosted languages: they talk the talk, but do they walk the walk? In *Proceedings of the 2013 International Conference on Principles and Practices of Programming on the Java Platform: Virtual Machines, Languages, and Tools*, pages 101–112. ACM, 2013.

[7] D. S. V. Sujala D Shetty. Interoperability issues seen in web services. *IJCSNS International Journal of Computer Science and Network Security*, 9:160–169, August 2009.

[8] Wikipedia. Common language runtime — wikipedia, the free encyclopedia, 2014. [Online; accessed 18-March-2014].

[9] Wikipedia. Comparison of java virtual machines — wikipedia, the free encyclopedia, 2014. [Online; accessed 25-March-2014].

[10] Wikipedia. Java virtual machine — wikipedia, the free encyclopedia, 2014. [Online; accessed 18-March-2014].

[11] Wikipedia. Just-in-time compilation — wikipedia, the free encyclopedia, 2014. [Online; accessed 18-March-2014].

[12] Wikipedia. Virtual machine — wikipedia, the free encyclopedia, 2014. [Online; accessed 18-March-2014].