Interoperability in Programming Languages

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

ABSTRACT

Interoperability of programming languages is the ability for two or more languages to interact as part of the same system. Frequently, this means passing messages and data between potentially very different languages. The differences in these languages pose a serious barrier to creating an interoperative system.

However, interoperability is important to many existing systems today, as particular programming languages have emerged to target particular problem domains. In particular, client/server architecture and many distributed computing systems utilize multiple languages for differing parts of their system.

As no single approach is likely to address all problems that could arise, several tools and approaches have emerged to address different aspects of cross language communication. Two broad categories of these tools are virtual machines and markup languages. These two tools are used concurrently in many systems today, reflecting their different strengths.

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, web browsers frequently perform some kind of interop. Information to be displayed on a web page is handled by a browser's interpretation of HTML, while any real-time aspect of the page will be handled by a language like JavaScript, again run from within the browser. If there is more computation to be done, or if a database needs to be consulted, the client, the browser, 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 server written in both Java and Clojure, where Java handles translation between the client and server while Clojure handles the rest of the server-side logic.

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 we 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, Sections 3 and 4 will 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.

2. INTEROPERABILITY

The primary importance of interoperability lies in the varying capabilities of different languages. Programming languages are optimized for particular tasks, and different languages perform better in different situations.

Some languages are constructed to tackled specific problems, but provide reduced support for others. Erlang, for instance, is one of the primary languages used in distributed computing. It was built to handle remote procedure calls (RPCs), and does so simply and powerfully. However, intensive string manipulation in Erlang can be a challenge: Strings are not first class types in Erlang, nor do they even have a distinct type [**?**]. Strings in Erlang are treated as

[1]kidblog.org

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

character arrays, and while there is a module for string manipulation, it is significantly more light-weight than many more recent languages, such as Java or Ruby .

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. This low-level control can be useful when dealing with systems with minimal memory or when access to device drivers is needed for things like GPU computation. However, these languages have high learning curves, and programs using them may have security concerns 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 is now done with scripting languages like Ruby or JavaScript. 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. For instance, fuzzy controllers, which in part translate real-world input to fuzzy sets, are often strictly tied to the hardware they will be implemented upon. FML, the Fuzzy Markup Language, was developed as a way to fully describe such a controller in a hardware-independent manner. Part of this hardware independence involves independence from any particular implementing language; as such, a fuzzy controller specified in FML is capable of being implemented in or potentially across several different languages. [1]

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.

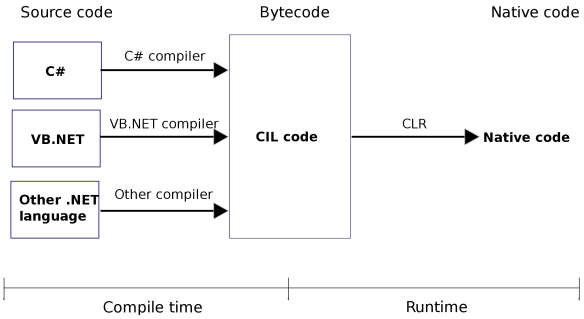

Figure 1: VM architecture in the CLR [7]

Specifically, they typically provide restricted, VM managed memory spaces, ensuring that hosted processes can't access memory beyond what is allocated to it. [11] is this relevant?

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.[2] 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. [10]

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 on 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 would be hosting that code. 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. Eventually, other languages began using the JVM for their runtime, but this was not the primary consideration of the original development team.

[2]In the CLR, this is actually called the Common Type System, or CTS. It is implicit and unnamed in the JVM.

[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. [8, 9]

```
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 (taken from [2])

In contrast, the CLR was developed to support 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 (MLs) 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 TEX is a language used to describe the layout and formatting of text documents. Others, such as JSON and 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 (see Fig. 2). Tags are easily distinguishable from the data they contain, and are considered separate from the data when the XML is read. In Figure 2, the fields (`Method, URI,` and `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. FML is one example of this, as are the MDLs in the Starlink framework.

Starlink [2] itself is much 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, which are network messages separated into their constituent parts and types. A translation logic specification, also written in XML, defines what parts in two MDLs can be related 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.

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

5. HANDLING INTEROP

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, consider a system where key/value map would be useful, but one of the languages involved 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.

For issues like these, there are two concepts that help enable this translation with minimal loss: metadata and standards.

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 [4]. Simply put, metadata is data describing data. This can be almost anything, as long as it conveys information beyond what the data can by itself. In the case of strongly-typed languages, the type information included as part of a vari-

```
Class Person
  string name = "Cliff"
  date dateOfBirth = 4/16/1978
  int height = 74
  double weight = 212
end
```

Figure 3: A Person class strongly-typed language

```
Class Person
  var name = "Cliff"
  var dateOfBirth = 4/16/1978
  var height = 74
  var weight = 212
end
```

Figure 4: A Person class in an untyped language

able declaration is a kind of metadata, as in Figure 3. There we can see that there is a difference in the representation of the `height` and `weight` fields, despite both fields being a number.

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.

As mentioned above, the explicit type information in strongly-typed languages, like Java or C# is a kind of metadata. This metadata is entirely for use by their compilers, to assist with translation to the intermediate language. In weakly or dynamically typed languages like Ruby and Groovy, this metadata does not exist (Figure 4). Since these languages perform run-time type checks instead of at compilation time, significantly more code is generated by their compilers to handle type checking, and they cannot take advantage of the intermediate language's primitive type system (everything must be assumed to be an Object-type object) 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, but can also be accessed by other tools.

MLs and metadata

Metadata is the strong point of markup languages, as they are designed around the same concept of describing data. Essentially, the fields in a markup language (tags, in the case of XML-based languages) act as metadata for the data they are attached to.

When translating into an object for another programming language,

The real metadata in a markup language can be stored in many ways. In Starlink's MDLs, for example, the type information of fields is stored in a `<type>` tag before the

```
<Person>
  <name> Cliff </name>
  <birthdate> 4/16/1978 </birthdate>
  <height> 74 </height>
  <weight> 212 </weight>
</Person>
```

Figure 5: An XML description of a person

definition of the fields begins, as can be seen in Figure 2 [2]. In other cases, the type information is stored in-line with specifications for correct formatting, as in XML schema (discussed in Section 5.2).

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.

There are two aspects to standards. One is agreement on similar data types, the other is the reaction to or approach to handling those types once received. Ide and Pustejovsky [4] refer to this difference as syntactic interoperability (agreement on data type and communication protocols) and semantic interoperability (the ability to act on data received in a way unsurprising to other components of a system). While syntactic interop can be handled in part by metadata, semantic interop can require knowledge of more detailed aspects of the components in a system.

Shetty and Vadivel [6] demonstrate an example of a failure in semantic interoperability even while retaining some (though not all) syntactic interoperability between Java and .NET (CLR languages) web services. While they found several syntactic mismatches, such as the lack of unsigned numbers in Java, there were issues even where the systems supported the same types. In particular, Shetty and Vadivel noted that null values were handled in largely incompatible ways. When given an array with a null element, Java interpreted as a null object, and printed `null` when that element was requested. In contrast, .NET interpreted the element as empty, and printed an empty string when the element was requested. Additionally, they found that precision of decimal, double, and float types differed between platforms. When asked to add and display 4.111111 and 8.999999, the Java client returned 12.999999 with full precision, while the .NET client returned 13.

Though these two services were handling the same data, represented by the same types, they behaved differently when asked to act upon that data. As Shetty and Vadivel point out, this has implications for clients who don't know which service architecture they will be interacting with[6].

VMs and Standardization

Although the above study found issues between virtual machines, systems run on homogeneous virtual machines architecture actually have an advantage. Systems on a virtual machine already have a standard implemented in the form of the intermediate language. Because the intermediate language has a full language specification, including both a type system and data interpretation behavior, it can act as a description for required behavior. Moreover, once a system has been compiled into the intermediate language, the behavior

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema elementFormDefault="qualified"
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="Person">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="name" type="xs:string" />
       <xs:element name="birthdate" type="xs:date" />
       <xs:element name="height" type="xs:double" />
       <xs:element name="weight" type="xs:double" />
     </xs:sequence>
   </xs:complexType>
 </xs:element>
</xs:schema>
```

Figure 6: A W3C-style XML schema for the person in Figure 6.

of the different components is constrained by that virtual machine's behavior.

This is not a perfect solution however, as it still requires that the high-level languages have conventions for accessing other high-level language components. These conventions are tied to compilers at least, so grammar for calling other languages on a particular VM can be included as part of a new compiler specification without affecting other implementations of that language.

As an example of this, Java was not initially intended to call other languages, and making calls to non-Java languages can be a difficult proposition [3]. However, new JVM languages like Clojure and JRuby inherently provide support for accessing Java classes. would this benefit from an explicit example?

MLs and Standardization

Semantic interop is somewhat more straight-forward to attain with markup languages. Although not built in, many markup languages have a notion of schema, such as in Fig. 7, which can be used to regulate correct formatting. Messages sent in the markup language can be checked against these schema to ensure both that they contain the right fields in the right places (ie, a nested field is contained by the correct super field), as well as that those fields contain the correct type of data. As Figure 7 shows, type metadata can be stored within the schema specifications.

It should be noted that schema and similar mechanisms are themselves simply a standard, and are not expressly enforced or singular. XML has several schema languages associated with it. Figure 7 is an example of the W3C[5] recommended standard.

6. PERFORMANCE

Regardless of how the system is built or the interop implemented, an interoperating system will always accrue some overhead. There may be unexpected costs from translating between languages, which can appear in places such as execution speed or program storage space

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 higher-level languages can be reasonably translated to the intermediate. Li, White, and Singer [5] 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 sequences of bytecode distinct from those produced from Java source code. For dynamically-typed languages like Ruby, these sequences are significantly longer than bytecode from an equivalent Java class. Additionally, JRuby bytecode contains none of the built-in type operators, which are used to optimize performance. This has some potential performance and space implications when comparing JRuby and Java.

While I am not aware of a similar study focusing on languages on the CLR, the CLR was designed both with interoperability in mind and with an awareness that dynamically-typed languages would be running on it. From these two points, it seems reasonable to assume that CIL bytecode and the CLR interpreter are designed to handle dynamic types more efficiently or concisely.

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, showed a non-negligible time lapse between when the initial message was received by Starlink and when Starlink returned a reply to that protocol [2].

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

Virtual machines and markup languages each have strengths and weaknesses in different use cases.

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.

In comparison, markup languages are better suited to dealing with preexisting or legacy systems, where there is too much code to make rewriting a feasible option, or where parts of the source are simply unavailable, potentially due to working with a third-party software. 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. This is particularly important in client/server architecture, where servers have few, if any, guarantees about the hardware or sometimes even the software being run by the clients. A markup language can act as an intermediary, providing the data in a language and hardware free way.

These two cases are not mutually exclusive, and can even be complementary. A real-world system is more likely to make use of both virtual machines and markup languages

[5]World Wide Web Consortium

to cover relevant aspects of their system than to rely on just a single approach.

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] 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.

[5] 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.

[6] 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.

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

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

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

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

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