System of 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

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Delphi theory

Keywords

ACM proceedings, LaTeX, text tagging

1. INTRODUCTION

Introduction will introduce what Interop is, what it means to be interoperable, and why it is important to people

What is Interoperability? Interoperability, often shortened to Interop, is the ability for two or more systems to work together. It is a very broad definition, describing anything from groups of people to bureaucratic systems to pieces of hardware. Here we will be discussing the interoperability of programming languages and the hardware used to run such programs.

First we will explore why interoperability is subject we are concerned with, as well as what is needed to achieve it and why it is so difficult to do so.

Next, we will take a look at two types of systems that help achieve interop. The first type will be virtual machines. What are virtual machines, and what about them makes them useful for this discussion? The Java Virtual Machine and the .NET framework provide examples.

Then we'll look at markup languages as a method of interoperability. Markup languages mesh nicely with the idea of metadata in interop, explored in section 2

UMM CSci Senior Seminar Conference, December 2013 Morris, MN.

2. INTEROPERABILITY

this section needs a better name

Why is is interoperability important in computing systems? The two main considerations here are what the languages themselves are designed to accomplish and what hardware the programs being built are meant to run on.

Programming languages have different purposes, and can be applicable in different situations. For instance, the language Erlang is very good at handling messages over distributed systems. But constructing certain data structures in that language can be frustrating. I remember this being a thing, but I don't remember what those structures were. Better check on that In contrast, Java has a nice suite of built in data structures, which allow for easier construction of further structures as needed. However, getting Java to pass synchronized messages over a network is not fun.

Thus, it would be useful if there were a way to pass messages across the network with Erlang, and use Java to handle some of the more heavy-duty data processing. This is probably a terrible example. What can we do to make this happen?

Requirements for Interop

In order for interop to be a viable option, the languages in question must have some common ground. At at basic level this means they should be able to act on similar data types and have access to the same types of files. However, simply dealing with data of the same type does not guarantee successful data sharing, as languages may deal with similar data differently. Ide and Pustejovsky [3] suggest a method for dealing with potential type conflicts through the use of metadata. Metadata can describe the type and classification of data in more broadly applicable and translatable fashion.

Of importance in every level of the transaction is standardization. Metadata, as mentioned, can be a useful tool for data interoperability. In order to actually use it, however, both languages or programs must agree on what tags will represent what data. Standardized metadata sets ensure that programs not built with each other in mind can still interoperate without being fully redesigned. There are several catalogs attempting to form full metadata tag sets, such as the Open Archives Initiative (OAI), the ELRA Universal Catalog, and the NICT Shachi catalog.[3]

Difficulties in achieving Interop

Despite tools available to promote interoperability, there are still difficulties inherent to building an interoperative system. Metadata tags can be a powerful too, but there are some subtleties about languages that should be taken into account.

One of these is the type systems of the languages in question. For instance, converting between the weakly-typed language Groovy and the strongly-type Java might require that all of the Groovy data be labeled, or it's arrays be of only a single type, effectively forcing Groovy into a strong type system. This requires some better introduction. also, look up how groovy does manage its Java interop

Another area of concern is adherence to standardization. As evidenced above, there exist a wide array of possible standards to use. Conformance to one standard does not guarantee connectivity with a system using a different standard, reducing the usefulness of such standards by the number of them in existence.[**?**]

3. VIRTUAL MACHINE SYSTEMS

One method of achieving interoperability is though the use of a virtual machine (VM). Virtual machines are programs that model a physical machine, capable of executing programs within their environment. Two major virtual machines in use today are the Java Virtual Machine (JVM) built primarily by Java, and the Common Language Runtime (CLR) used by Microsoft's .NET Framework.

Part of their power in interoperability comes from their method of execution. Unlike purely compiled languages where code is compiled down to machine code, or interpreted languages where code is converted directly to machine-level instructions, languages that run on the JVM or the CLR are compiled to an intermediate, low-level language, which is then translated to machine-level instructions. The intermediate languages are bytecode (Java) and the Common Intermediate Language (.NET)

This has two major implications: one for the interoperability of languages built to run on the same virtual machine, the other for the portability across platforms for languages built on a virtual machine.

Single Platform Language Interoperability

Virtual machines can be used as a language interoperability platform when the system in question is meant to be run on a single machine. When used like this, the common low level language provides the intermediary between languages. Here we can see the usefulness of standardization as described by Ide and Pustejovkey[3].

Languages designed to run on a virtual machine have access to standardized libraries via a commonly accessible language. For the JVM, this is Java itself and the Java standard libraries. For the .NET framework, I have no idea how they're implemented [1] with the Base Class Library.

The .NET framework has an advantage over the JVM in terms of standards. When the JVM was designed, it was designed as a platform only for Java, and thus didn't take into consideration that there could be other languages running on it. However, .NET was built from the ground up as a suite for multiple languages, and thus incorporated cross-language standards early in its life.

Regardless, both have a core language and shared data types. Languages on both have access to methods and objects in other languages on the same system. This is more difficult on the JVM, as most languages likely have to go through Java to achieve this, while .NET languages have nearly direct access.not quite that much. The common compiled language (bytecode and CIL) means that once a program utilizing multiple languages has been written and compiled, execution of all parts of the program can be handled the same way.

[1]seriously, is this a language here? Does it conform to a language used?

Limitations

Of course, this is only of practical use if program in question is simple enough to be run on a single machine. In practice, multi-language systems are much more likely to happen on a distributed scale, where different languages handle different stages of the project.

Platform Portability

A second useful area of virtual machines is the ability to separate the language from hardware. This can be useful for interoperability if the same program is needed to run on multiple machines, which may not have the same underlying hardware.

This approach relies on the fact the the virtual machine is an abstraction of the base hardware. Any program will need some sort of translation to go from the high-level user code to the hardware-specific machine code at execution time. VMs offer the advantage of modeling the underlying hardware while allowing higher-level programmers to ignore that hardware. Platform specific implementations allow the underlying system to change without impacting deployed programs, and allows programs to change without having to worry about the hardware. I just said that three times. Yuck

This is actually the purpose the JVM was designed for, and one of the primary reasons existing languages created JVM implementations. As Li et al. discovered, this meant building in interoperability with Java, as Java executed better on the JVM, being native, and provided much of the functionality that the foreign language would then not have to duplicate[4].

Limitations

Being able to run your program on multiple machines is one thing, but having that

4. MARKUP LANGUAGE SOLUTIONS

Markup languages, such as XML or HTML, are at their core a method for adding metadata to text via tags. This makes them an almost perfect match for Ide and Pustejovsky's model of metadata language description. ML tags can be added to any string data, which can be very handy for shuttling data around.

perhaps describe different base types of MLs here? I'm not sure

Distributed Connectivity

Markup languages are primarily useful when dealing with systems that span multiple servers or platforms. Need more here

Markup languages, by their heavily tagged structure, [2] are a perfect match for Ide and Pustejovsky's model of metadata

[2]EXPLAIN THE HECK OUT OF THIS

interoperability.[3] The tagging system used to describe aspects of a markup language can be adapted to describe data types within other languages and data models.

One system that makes use of this is the Starlink framework[2]. Starlink's purpose is to facilitate passing messages between languages that do not share a common communication protocol. As such, for each language it must model both the incoming message and the protocols for each language. The translation of the message between representations is handled using sets of logical automata, while the representations themselves are done using a specialized markup language they call the Message Description Language. In an MDL, each piece of data is contained within its own set of ML tags , referred to as primitive fields. Primitive fields are themselves contained in another set of tags, called structured fields. These fields can be likened to the metadata labels, with primitive fields standing for syntactic level labels and structured fields standing for semantic level labels. I'm not sure that's strictly true, so look into this. Consider including figure 7 of [2] here

Consider using: MDLs are set up in a specific way so that the automata are able to efficiently translate between two MDL representations

FML and Hardware Independence

Markup languages can also be used to divorce programs from hardware dependencies, freeing them to be more interoperable over distributed platforms.

Will need some description of what's nice about fuzzy controllers The Fuzzy Markup Language (FML) is a prime example of this. At present, the best method for implementing fuzzy controllers for a particular system has involved designing specific hardware models based on the system it was being designed for. When dealing with a number of platforms using different hardware, this meant different designs were required to implement the same controller for each. FML attempts to shift the control of fuzzy logic from hardware to software, such that it can be implemented on multiple platforms for little additional cost.

Consider using [1] figure 1 to explain the fuzzy system Once again, While significantly more complicated, FML can be described in a basic sense to be using the XML tags as a means of attaching metadata to data involved in a fuzzy control system. It is more complicated because they use more than the two levels of metadata described by Ide and Pustejovsky. FML is meant to support a fairly complex system, and the models used within that system for determining stuff. As such, they go beyond modeling primitive data to modeling full data structures. It goes beyond this, but out of the realm of metadata.

5. CONCLUSIONS

6. ACKNOWLEDGMENTS

7. 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. *One example of an ML system. May be useful for comparison of ML style interop.*

[2] Y. Bromberg, P. Grace, and L. ReÌĄveilleÌĂre. Starlink: Runtime interoperability between heterogeneous middleware protocols. In *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, pages 446–455, 2011. *An intensive exploration of what is required to achieve interop. Describes a full interop system using an ML.*

[3] 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. *A general exploration of the definitions and requirements of interoperability.*

[4] 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. *Explores how languages on the JVM differ from Java.*