PL/I for OpenVMS and Tru64

WSIT and J2VMS: A Comparative Analysis

Tim E. Sneddon

Introduction

The purpose of this article is to compare the two products:

- HP's Web Services Integration Toolkit; and
- Kednos' Java-to-VMS Toolkit.

These two products will, from now on, be referred to as WSIT and J2VMS, respectively. In starting I should probably state that although I have attempted to remain somewhat objective, I am biased. Considering that I am the current developer/maintainer of J2VMS I do have a preference. That said, I am also an advocate of the belief that one should use the best tool for the job. So in this article I intend to pick out the strengths and weaknesses of both products.

Each product presents a method of calling native routines and manipulating native data structures from within Java. However, the one fundamental difference is that WSIT has been designed with the idea of native code being directly called from some sort of web service. J2VMS aims to exist outside of that restriction and provide a toolkit for doing much more.

So with that in mind I will begin by first looking at how both products present and manipulate data structures, followed by the declaration and use of constants and finally, the native calling interface.

Acknowledgements

I would just like to acknowledge the WSIT developers and documenters. In some cases I have borrowed their examples for this comparison. Also, Jim Brankin, the original author of J2VMS.

Structures

Both WSIT and J2VMS support the concept of structures. However, they differ quite a bit in the way they are presented to the developer. In both instances it is necessary to describe the structures in a language other than Java. WSIT describes structures using IDL and J2VMS uses SDL. Although it is not strictly necessary to use SDL, it is highly recommended.

For this example I will use the sellerData structure from the WSIT 'stock' sample. The example below shows the C declaration of this structure.

```
const MAX_STRING = 20;

typedef struct _sellerData {
        char owner_name[MAX_STRING];
        unsigned int member_number;
        unsigned int balance_dollars;
        unsigned int number_shares_available;
} sellerData;
```

The snippet below shows the equivalent declarations in SDL for J2VMS. In this context SDL refers to the Structure Definition Language. It is a language and compiler for generating include files for many language variants from the one source file.

```
constant MAX_STRING equals 20;

aggregate sellerData structure typedef tag "" prefix"";
    owner_name character length MAX_STRING;
    member_number longword unsigned;
    balance_dollars longword unsigned;
    number_shares_available longword unsigned;
end sellerData;
```


A benefit of SDL is the language supports a number of primitive data types that cover those usually found in a high level language. This is a feature not present in the IDL equivalent, which can be seen in the following example. It is required that all primitive data types are declared, as well as complex types. The following IDL example has been generated using the OB2IDL tool.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenVMSInterface …>
  <Primitives>
    <Primitive Name = "unsigned int"
               Size = "4"
               VMSDataType = "DSC$K_DTYPE_LU"/>
    <Primitive Name = "AutoGen_FixedString19"
               Size = "19"
               VMSDataType = "DSC$K_DTYPE_T"
               NullTerminatedFlag = "0"
               FixedFlag = "1"/>
  </Primitives>
  <Typedefs>
    <Typedef Name = "sellerData"
             TargetName ="_sellerData"/>
  </Typedefs>
  <Structure Name = "_sellerData"
             TotalPaddedSize = "32">
    <Field Name = "owner_name"
           Type = "AutoGen_FixedString19"
           Offset = "0"/>
    <Field Name = "member_number"
           Type = "unsigned int"
           Offset = "20"/>
    <Field Name = "balance_dollars"
           Type = "unsigned int"
           Offset = "24"/>
    <Field Name = "number_shares_available"
           Type = "unsigned int"
           Offset = "28"/>
  </Structure>
</OpenVMSInterface>
```

IDL stands for Interface Definition Language. It is an XML formatted language that describes the structures, constants, data types and the call interface to the WSIT code generator. Unfortunately there are no tools to use these same declarations with other languages and the developer is required to maintain multiple declarations of the same structures. Re-generating these declarations can also become difficult if OBJ2IDL is unable to resolve a data type itself. One advantage that IDL does have over SDL is the built-in support for multi-dimensional arrays. SDL does not currently support this feature.

The following extract is the code generated by SDL that describes the data structure, sellerData, to J2VMS.

```
public class stockdef {
  public static final int MAX_STRING = 20;
  public static final FieldDescriptor owner_name = new FieldDescriptor(0,0,0,0);
  public static final int s_owner_name = 20;
  public static final FieldDescriptor member_number = new FieldDescriptor(20,0,32,1);
  public static final FieldDescriptor balanace_dollars = new FieldDescriptor(24,0,32,1);
  public static final FieldDescriptor number_shares_available = new FieldDescriptor(28,0,32,1);
  public static final int s_sellerData = 32;
}
```

It shows each of the fields as a descriptor detailing the field's position, size and signedness in terms of bits and bytes as offset from the base of the structure. This is a concept borrowed from the BLISS programming language. J2VMS presents all structures to the developer using the `vs.VmsStruct` class. It provides a collection of methods for reading and writing the data structure which, internally to `vs.VmsStruct`, is simply an array of bytes. These field descriptors provide an address into the byte array making it possible for

the developer to gain direct access to the data structure to fetch or store any value necessary.

This implementation also allows for a far more generic interface. The only code that is generated is the class containing the field definitions. It can be thought of as the equivalent of a header file in a compiled language like PL/I or C. The class `vs.VmsStruct` is part of the J2VMS class library.

The following extract shows a sample of the code generated by WSIT. Each structure is represented as a single class and as such requires that a separate file be generated for each one.

```
/**
 * _sellerData
 *  This class represents the _sellerData structure.   It contains all
 *  of the accessor functions for the individual fields of this structure.
 *  <p>
 *  Please be aware that the encapsulated class, _sellerDataImpl, contains
 *  all of serialization methods needed for this class, and are for internal
 *  use only.
 */
public class _sellerData implements WsiV2Structure {
    …
   /**
     * getOwner_name()
     *    returns the owner_name value for this _sellerData.
     *
     * @return the owner_name field of this structure
     */
    public String getOwner_name() {
        return wsiInnerStruct.getOwner_name();
    } // getOwner_name

    /**
     * setOwner_name()
     *
     *  WARNING: This method should not be used by WSIT customers.
     *  For Internal use only. To populate a structure field first
     *  use the getOwner_name method to obtain a reference
     *  then populate the returned object.
     *
     *  Sets the owner_name value for this _sellerData.
     *
     * @param owner_name        New value for the owner_name field.
     */
    public void setOwner_name (String owner_name ) {
        wsiInnerStruct.setOwner_name(owner_name);
    } // setOwner_name
    …
}
```

Due to the requirement that WSIT generate classes that conform to the JavaBean convention, all fields are represented as getter and setter methods. These methods in turn manipulate private members that contain the stored value. This convention also prevents the developer from creating structures that contain pointers to other structures or memory regions. This can make it impossible to call code that requires access to linked lists or tree structures.

The following two examples demonstrate how to load and fetch the owner_name field in the sellerData structure. The first uses the WSIT environment.

```
_sellerData sellerData = new _sellerData();

sellerData.setOwner_name("Hewitt Packard");
String owner_name = sellerData.getOwner_name();
```

The second uses J2VMS.

```
sellerData = new vs.VmsStruct(stockdef.s_sellerData);

sellerData.put(stockdef.owner_name, stockdef.s_owner_name, "Kedos Enterprises");
String owner_name = new String(sellerData.get(stockdef.owner_name, stockdef.s_owner_name);
```

Constants

Constants are handled very differently in WSIT and J2VMS. Declaring a constant in the WSIT IDL requires a special block of code that declares an enumeration and its values. Where as using the SDL language to define the constants available to J2VMS is simply a single declaration per constant. The following examples demonstrate the difference in complexity between the two methods.

First the constants are declared in their respective languages.

WSIT definition in IDL	J2VMS definition in SDL
<?xml version="1.0" encoding="UTF-9"?> <OpenVMSInterface….> <Enumerations> <Enumeration Name="TES_ANSWERS" ByteSize="4"> <Enumerator Name="TES_TRUE" ConstantValue="1"/> <Enumerator Name="TES_FALSE" ConstantValue="0"/> </Enumerations> </Enumerations> </OpenVMSInterface>	module TESDEF; constant TES_TRUE equals 1; constant TES_FALSE equals 0; end_module TESDEF;

Once the respective code generators have been run the developer is presented with a Java class for each. In the case of WSIT it is the JavaBean interface class. The following extract shows the code generated for the IDL declaration above.

```
/**
 * This class represents the TES_ANSWERS Enum type.
 */
public final class TES_ANSWERS implements java.io.Serializable
{
    private int m_value;
    /**
     * This property contains the value of TES_TRUE within the TES_ANSWERS Enum type.
     */
    public static final TES_ANSWERS TES_TRUE = new TES_ANSWERS(1);
    /**
     * This property contains the value of TES_FALSE within the TES_ANSWERS Enum type.
     */
    public static final TES_ANSWERS TES_FALSE = new TES_ANSWERS(0);

    /**
     * Creates and initializes a TES_ANSWERS object with a given value.
     */
    private TES_ANSWERS (int value)
    {
        m_value = value;
    }

    /**
     * getValue() returns the value assigned to this enum.
     */
    public int getValue()
    {
        return m_value;
    }
…
}
```

Although they have been left out of the extract above, WSIT also generates `toString` and `equals` methods. Where as SDL generates the following extract for use with J2VMS:

```
public class TESDEF {
  public static final int TES_TRUE = 1;
  public static final int TES_FALSE = 0;
}
```

As can be seen here, J2VMS does not create a whole class for accessing a named constant. Instead it presents a read-only value that the developer can then use as they see fit, avoiding all the extra work associated with instantiation and using getter and setter methods to access a constant value. SDL also supports the declaration of named string constants.

It is also worth noting that it is not possible to detect constant values in objects using the OBJ2IDL utility. These need to be added by hand.

Routines

J2VMS and WSIT have rather different ways of supporting calls to native routines. This is mostly due to the distributed nature of WSIT. J2VMS can certainly be called by a web service, but a wrapper method needs to be written by the developer. This does have the advantages of hiding several calls behind the one web service as well as being useable from any type of Java web service.

To demonstrate the differences between calling mechanisms in WSIT and J2VMS the 'math' sample that is packaged with WSIT will be used. The following snippet shows the C function prototypes for the routines prod and sum.

```
unsigned int sum ( int number1, int number2);
unsigned int product ( int number1, int number2);
```

Using the WSIT utility OBJ2IDL, it generates an IDL description similar to the following extract. Only the sum routine is shown as both routines share the same arguments and return values.

```
<?xml version="1.0" encoding="UTF-8"?>
<OpenVMSInterface …>
  <Primitives>
    <Primitive Name = "unsigned int"
               Size = "4"
               VMSDataType =
"DSC$K_DTYPE_LU"/>
    <Primitive Name = "signed int"
               Size = "4"
               VMSDataType = "DSC$K_DTYPE_L"/>
  </Primitives>
  <Routines>
    <Routine Name = "sum"
             ReturnType = "unsigned int">
      <Parameter Name = "number1"
                 Type = "signed int"
                 PassingMechanism = "Value"
                 Usage = "IN"/>
      <Parameter Name = "number2"
                 Type = "signed int"
                 PassingMechanism = "Value"
                 Usage = "IN"/>
    </Routine>
  </Routines>
</OpenVMSInterface>
```

In this example it can be seen that, all primitive data types must again be declared and the argument types and mechanisms are set. An issue that I noted when using OBJ2IDL against a BASIC program was its inability to recognize a dynamic string, the only string

format used by BASIC for automatically declared strings. This is yet another instance that requires human intervention.

Once the developer has a correct IDL representation of their routines it is necessary to generate the code that does the work. This is done using the IDL2CODE tool. This tool generates a collection of files that represent the server-side wrapper and the client interface. All up a total of 10 files for the prod and sum functions. These include build procedures, native C code for a shareable image used to call the native routines plus Java classes to expose the native routines to the Java/WSIT environment.

Displaying code snippets from all these files would not be a very productive use of space, so I will skip that. However, I do encourage the reader to investigate this for themselves. The samples that ship with WSIT can be generated with relative ease.

Using the auto-generated code it is now possible to write a class that calls the native routine. This can be done either "in-process" or "out-of-process". In-process refers to a local call. Meaning that the native call occurs in the context (and address space) of the running process. This type of call looks something like this:

```
import math.*;
import java.io.*;

public class mathcaller {
    /** Creates a new instance of Main */
    public mathcaller() {
    }

    public static void main(String[] args) {
        try {
            mathImpl math = new mathImpl();
            int result;

            result = math.sum(10, 15);
            System.out.println("Result = " + result);
        } catch (Exception e) {
            System.out.println("Exception thrown");
            e.printStackTrace();
        }
    }
}
```

Out-of-process refers to the web service call. This is done by calling the routine using the Java Remote Method Invocation (RMI) interface. It looks something like this:

```
import math.*;
import java.io.*;
import com.hp.wsi.WsiIpcContext;

public class mathcaller {
    /** Creates a new instance of Main */
    public mathcaller() {
    }

    public static void main(String[] args) {
        try {
            mathImpl math = new mathImpl(new WsiIpcContext());
            int result;

            result = math.sum(10, 15);
            System.out.println("Result = " + result);
        } catch (Exception e) {
            System.out.println("Exception thrown");
            e.printStackTrace();
        }
    }
}
```

In the case of J2VMS there are two ways that these routine can be declared and called and in both instances the call is "in-process". The first is similar to that of a header file. SDL is used to declare the functions and generate a class that can then be used to call these functions. The following SDL shows the declaration for sum.

```
module math;

entry sum parameter(
    longword unsigned named number1 in  value,
    longword unsigned named number2 in  value)
  returns longword unsigned;

end_module math;
```

SDL is then used to generate a class containing the following routine declaration. (It is assumed, for this example, that the sum routine is contained within the MATH_RTL shareable image.)

```
package org.tes;
import vs.VMSparam;
import vs.SystemCall;
import vs.FieldDescriptor;

public class MATH { // IDENT
  private static SystemCall nullclass;
  private static final String libname = "MATH_RTL";
  private static SystemCall sum_return;
  public static int sum(VMSparam[] args) {
    if (sum_return == nullclass) {
      sum_return = new SystemCall("SUM",libname);
    }
    return sum_return.call(args);
  }
}
```

This then allows the developer to import the class into their Java program and call the routine like so.

```
Import java.io.*;
import org.tes.MATH;

public class mathtest {
    private MATH math = new MATH();

    public static void main(String[] args) {
        int result = math.sum(new VMSparams[] {
                                    new ByValue(10),
                                    new ByValue(15)
                                });
        System.out.println("Result = " + result);
    }
}
```

Incidentally this is the method that J2VMS uses to provide its own version of the STARLET libraries. This allows the developer to easily call their favourite LIB$ or SYS$ routines.

The second, and simpler method, is much like adding a single external routine reference in a native high level language. This is an advantage for simple programs that require maybe one or two functions. This allows the developer to quickly declare the routine without having to go down the path of generating a header file and adding it to their build environment. This method is demonstrated in the following example.

```
public class mathtest2 {
    public static void main(String[] args) {
        SystemCall sum = new SystemCall("SUM", "MATH_RTL");

        sum.call(new VMSparams[] {
                    new ByValue(10),
                    new ByValue(15)
                });
    }
}
```

Both of these examples show that the developer has ultimate control over the argument passing mechanism. This is a feature that does not appear to be supported by WSIT which locks the developer in with the wrapper code it generates. J2VMS passes all arguments using the `vs.VMSparams` class. This class acts as a homed arguments list. This allows the `vs.SystemCall` method `call` to work in a very similar manner to LIB$CALLG.

Using the J2VMS environment it is not possible to automatically generate an "out-of-process" call method. It is left up to the developer to write the web service wrapper. This is largely done because it leaves the developer free to utilize the native call interface exactly how they want to. By writing their own wrappers, developers can include multiple calls in one web service call. They can also hide some of the obscurity of native calls that does not fit well into some of the web service interface standards available. It is certainly not possible to build something like an item list using WSIT. However, this is something that J2VMS has no problem with.

Conclusion

In conclusion I believe that both WSIT and J2VMS have their place. If you're looking for a quick Java RMI service, then WSIT is the way to go. However, J2VMS allows the developer to accomplish so much more. J2VMS allows developers to easily code calls to their favourite system services and run-time libraries to work with any type of web service they like. Be it Java RMI, SOAP or a simple HTTP servlet. In fact, just a plain old Java application is easily done as well. J2VMS can be used to easily develop software that fits all these environments. There is also the added bonus that the SDL files used to describe the native structures and routines for use with J2VMS are also perfectly useable with any other language supported by SDL (almost all native language, if not all). This keeps everything in the one place and reduces mistakes when updating many files describing the same thing.

Also worth mentioning is that all software required to use J2VMS runs on both OpenVMS Alpha and I64 (and for the really depraved, the SDL backend also runs on OpenVMS VAX). There is no need to ship objects to an OpenVMS I64 system to generate interface definitions, as there is with WSIT and the OBJ2IDL tool which only runs on OpenVMS I64. J2VMS is also much smaller, consisting of a shareable image and a Java archive.

I guess the last thing I should mention is that some of you may have noticed that I have avoided the topic of ACMS entirely. This has been deliberate. I do not have a lot of experience with ACMS and so don't feel in a position to make comment on the usefulness of WSIT or J2VMS in that sort of environment.

For more information

Tim Sneddon can be contacted via email at tsneddon@kednos.com.

The software and documentation for HP's Web Services Integration Toolkit (WSIT) can be found at the following website:

- http://h71000.www7.hp.com/openvms/products/ips/wsit/index.html

All software and documentation (including an evaluation license) for Kednos' Java-to-VMS Toolkit (J2VMS) can be found at this website:

- http://www.kednos.com/kednos/Integration/Java

Details of the SDL language and the SDLEXT collection of backends can be found here:

- http://www.kednos.com/kednos/Integration/SDL

Further details of the JavaBean conventions can be found at the following resources:

- Wikipedia, *JavaBean*
  - http://en.wikipedia.org/wiki/JavaBean
- Sun, *Java SE Desktop Technologies – JavaBeans*
  - http://java.sun.com/javase/technologies/desktop/javabeans/index.jsp