The Vulcan Bridge

by John Ellis, Bill Kalsow, Eric Roberts, Michael Sclafani

Last modified on Thu Aug 30 09:44:58 1990 by kalsow

August 30, 1990

Contents

1. Introduction

This document describes the Vulcan bridge. Before reading this document, you should be familiar with Vesta and the Vesta language.

The Vulcan bridge defines a set of primitive functions. The functions defined by the bridge and described below are: `Compile`, `Bind`, `Make-program`, `Make-load-and-go`, `Make-shared-library`, `Make-dynamic-library`, `Bag`, `Foreign`, `ToBinding`, `Flume`, `Caulk`, `Rename`, `Analyze` and `Print`.

The standard Vulcan model in the `vubasement` package defines additional values and functions that are described below. The additional values include `Prog`, `Load-and-go`, `Lib`, `Dyn` and default values for the options of these functions.

The bridge functions, primitive and otherwise, operate on values in the Vesta value space and repository. These values include sources, code modules, ASTs, objects, interfaces, implementations, executables, programs, shared libraries, and dynamic libraries. A *source* is a text string containing a Modula-2+ module. A *code module* is the machine instructions and tables produced by a compilation much like a Unix `.o` file (see `VuObject.def`). An *AST* is the abstract syntax tree produced by a compilation (see `M2AST.def`). An *object* is the Vesta value that denotes the ¡AST, code module¿ pair that results from compiling a Modula-2+ implementation or program module. An *interface* is the Vesta value that denotes the AST that results from compiling a Modula-2+ definition module. An *implementation* is the Vesta value that denotes an interface, an object, program or library that implements that interface, and references to actual implementations for each of that object's imports. An *executable* is the executable code returned when `Make-program` is applied to a main program's implementation. A *program* is the Vesta value that denotes an executable. A *shared library* is a collection of implementations that may be loaded into Vulcan's shared library space. A *dynamic library* is a collection of implementations that may be loaded into a running Vulcan address space. Sources, code modules, ASTs, executables, shared libraries and dynamic libraries are stored in the Vesta repository.

The Vulcan bridge functions check the `M2-verbose` flag. If it is set to `TRUE`, they produce diagnostic information as they operate.

2. Compile

```
Compile: (source          : text *
          M2-profiling    : bool *
          M2-checking     : bool *
          M2-incremental  : bool *
```

```
        M2-optimization : int *
        M2-is-basis     : bool *
        ... ) -> binding
```

Compile invokes the Vulcan compiler on the Modula-2+ module source. The compiler attempts to compile source. If no compilation errors are detected, it produces an AST (Abstract Syntax Tree) and possibly a code module which are stored in the Vesta repository. Finally, the Compile function returns a single-element Vesta binding whose name is derived from the source module's and whose value is either an interface or an object.

The first thing that Compile does is locate the interfaces imported by the source module. There are four ways that the source may specify an imported interface:

```
IMPLEMENTATION MODULE A ...

MODULE X IMPLEMENTS A ...

IMPORT A ...

FROM A IMPORT ...
```

In each of these cases the compiler looks for a value with the Vesta name A.d. These values must denote interfaces. Compile searches the environment enclosing the call for the imported interfaces.

In addition to the interfaces explicitly specified in the source, every module implicitly imports M2Base. M2Base defines the builtin types and procedures required by the compiler. The same search is used to locate M2Base.d as for any other interface. For all but wizards, the standard model in the vubasement package defines an appropriate interface M2Base.d.

If source is a definition module, the derived name in the result binding is the module's name concatenated with ".d". The resulting value is an interface. For example, a definition module that begins:

```
DEFINITION MODULE Foo;
  IMPORT Bar, Glarch;
  ...
```

will return the binding { Foo.d ~ <interface Foo> }. Compiling Foo will require interfaces M2Base.d, Bar.d and Glarch.d.

If source is an implementation module, the derived name is the module's name concatenated with ".o". The resulting value is an object. Compiling

```
MODULE X IMPLEMENTS A, B;
  IMPORT Text, Wr;
  ...
```

results in the binding `{ X.o ~ <object> }`, where `<object>` denotes the AST and code module produced by the compilation. Compiling `X` will require interfaces `M2Base.d`, `A.d`, `B.d`, `Text.d` and `Wr.d`.

Compiling a program module is the same as compiling an implementation module. For example,

```
MODULE Main;
  IMPORT Text, Wr;
  ...
```

results in the binding `{ Main.o ~ <object> }`, where `<object>` denotes the AST and code module produced by the compilation. Compiling `Main` will require interfaces `M2Base.d`, `Text.d` and `Wr.d`.

`Compile` recognizes several "options" (i.e. parameters with default values provided by the `vubasement` package's model). The options and their default values are:

`M2-profiling`	`FALSE`
`M2-checking`	`TRUE`
`M2-incremental`	`TRUE`
`M2-optimization`	`0`
`M2-isBasis`	`FALSE`

If `M2-profiling` is true, code is generated to collect runtime profile information. If `M2-checking` is false, no code is generated to detect runtime errors. If `M2-incremental` is false, no attempt is made to reuse the results of existing compilations of the same module. Currently, the `M2-optimization` switch is ignored. `M2-isBasis` is a "wizards-only" switch. If it is true, `Compile` doesn't add `M2Base.d` to the list of imports.

3. Bind

```
Bind: (pieces: list * ... ) -> binding
```

Given a list of objects or bindings, `pieces`, `Bind` searches the environment for implementations that satisfy the imports of those objects, binds the objects to those implementations and returns a Vesta binding containing implementations for the interfaces exported by the supplied objects.

For each object `A.o` passed to `Bind` and each import `B` in that object, `Bind` searches the environment for the name `B.i`. `B.i` must be bound to an implementation of the interface `B.d` that was used when `A.o` was compiled. If `B.i` is not found or is not bound to a proper implementation, an error is generated. Any import that can be satisfied by one of the implementations returned by `Bind`, is bound to that implementation. That is, imports are satisfied by first searching the exports of the actuals passed to `Bind` and then the enclosing Vesta environment. Hence, recursive imports must be bound by passing all objects involved in the recursion to a single invocation of `Bind`.

The Vesta binding returned by `Bind` contains an element

```
E.i ~ <implementation>
```

for each interface `E.d` exported by an object passed to `Bind`. The name `E.i` is derived from the name that occurs in the exporting module. For example,

```
Bind ((Compile (A.mod)))
```

where `A.mod` begins `IMPLEMENTATION MODULE A...` will return the binding

```
{ A.i ~ <implementation> },
```

.

If `pieces` includes bindings that refer to objects, it is as if just the objects were passed; the names in the binding are ignored. That is,

```
Bind (( { X ~ <object1>, Y ~ <object2> } ))
```

is interpreted as

```
LET { z ~ { X ~ <object1>, Y ~ <object2> }} IN
   Bind (( z$X, z$Y ))
END-LET.
```

It is an error for `pieces` to include bindings that refer to anything other than objects.

The most common uses of `Bind` will resemble

```
Bind (( Compile (A.mod),
        Compile (B.mod) ))
```

where `A.mod` and `B.mod` begin

```
IMPLEMENTATION MODULE A; ...
```

and

```
IMPLEMENTATION MODULE B; ...
```

This application reduces to

```
Bind (( { A.o ~ <object1> },
        { B.o ~ <object2> } ))
```

after the compilations and finally to

```
{ A.i ~ <implementation of A>,
  B.i ~ <implementation of B> }.
```

When Bind is applied to the object resulting from the compilation of a program module, it returns a binding where the name is the name of the program module and the value is its implementation (although strictly speaking a program module doesn't implement an interface and hence can't produce an implementation). So, if Main.mod begins MODULE Main; ...,

```
Bind (( Compile (Main.mod) ))
```

will return

```
{ Main ~ <implementation> }.
```

4. Make-program

```
Make-program: (pieces      : binding *
               M2-boot     : text *
               M2-startup  : impl *
               ... ) -> binding
```

Make-program collects the implementations required to build an executable Unix a.out file and returns a binding that describes the resulting program. The collection of implementations includes the main program's implementation and transitively any implementations imported by the collection. Implementations that reside in shared libraries are not copied into the resulting program; they are loaded into shared library space if necessary when the program is run. It is an error if any of the implementations in the collection are in other executables or dynamic libraries.

pieces is a binding whose elements refer to implementations. Exactly one of the implementations in pieces must be the implementation of a program module. Implementations other than the program module's are allowed in pieces so that dynamic libraries can refer to those implementations. When a dynamic library that uses implementations in an executable is loaded into an address space containing that executable, its imports resolve to implementations in the executable. The returned binding defines the same set of names as in pieces but rebinds them to the implementations in the resulting executable.

M2-boot contains the a.out header and boot code necessary to start executing the program. The boot code determines whether the resulting executable uses Taos's shared library support or can run on a standard Unix system. An exectuable that imports implementations from shared libraries, but runs without the shared library support, will abort when executed. Appropriate values for M2-boot are defined by the vubasement package's model: Vulcan-taos-boot is the default value, Vulcan-unix-boot is an alternative that runs on standard Unix.

M2-startup is a specialized implementation module that the boot code invokes to start the program. An appropriate value for M2-startup is defined by the vubasement package's model.

The buildingenv *model should also define a value for* M2-startup *that can be shared among address spaces.*

5. Prog

Prog is a short-cut that applies Make-program to the result of Bind. Its definition is:

```
Prog ~ LAMBDA pieces, M2-boot, M2-startup, ... IN
          Make-program (Bind (pieces), M2-boot, M2-startup)
       END-LAMBDA
```

6. Make-load-and-go

```
Make-load-and-go: (pieces     : binding *
                   M2-loader : text *
                   ... ) -> binding
```

Make-load-and-go is just like Make-program except that it builds a "load-and-go" executable rather than a Unix a.out file.

pieces is as in Make-program.

M2-loader is a specialized program that knows how to load, link and execute the program resulting from Make-load-and-go. The resulting executable can access shared libraries if and only if the loader can. Appropriate values for M2-loader are defined by the vubasement package's model: Vulcan-taos-loader is the default value which uses shared libraries, Vulcan-unix-loader is an alternative that runs on standard Unix.

7. Load-and-go

Load-and-go is a short-cut that applies Make-load-and-go to the result of Bind. Its definition is:

```
Load-and-go ~ LAMBDA pieces, M2-loader, ... IN
                 Make-load-and-go (Bind (pieces), M2-loader)
              END-LAMBDA
```

8. Make-shared-library

```
Make-shared-library: (pieces: binding * ... ) -> binding
```

Make-shared-library is like Make-program except that it builds a shared library instead of an executable. Shared libraries are loaded into Vulcan's "shared library space" on demand. Once loaded a shared library is marked "copy-on-write". Thereafter, unmodified pages of the library are shared among all of its clients on a single machine.

pieces is a binding whose elements refer to implementations. The returned binding defines the same set of names as the input binding but rebinds them to the implementations in the resulting library.

Like Make-program, Make-shared-library collects those implementations given in pieces and transitively any implementations imported by the collection. Implementations that already reside in shared libraries are not copied into the resulting library; they are loaded into shared library space if necessary when the program is run. It is an error if any of the collected implementations are in dynamic libraries or executables or implement program modules.

9. Lib

Lib is a short-cut that applies Make-shared-library to the result of Bind. Its definition is:

```
Lib ~ LAMBDA pieces, ... IN
        Make-shared-library (Bind (pieces))
      END-LAMBDA
```

10. Make-dynamic-library

```
Make-dynamic-library: (pieces: binding * ... ) -> binding
```

Make-dynamic-library is like Make-shared-library except that it builds a dynamic library instead of a shareable one. Dynamic libraries can be loaded into running Vulcan address spaces. Dynamic libraries may depend on (i.e. import) interfaces contained in programs and other dynamic libraries while shared libraries cannot. A dynamic library can depend on at most one program. Dynamic libraries cannot be loaded into shared library space and are not loaded on demand. It is up to the client to ensure that dynamic libraries are loaded as needed (see VuImage.Load and VuImage.Link).

pieces is a binding whose elements refer to implementations. The returned binding defines the same set of names as the input binding but rebinds them to the implementations in the resulting library.

Like Make-shared-library, Make-dynamic-library collects those implementations given in pieces and transitively any implementations imported by the collection. Implementations that already reside in shared libraries are not copied into the resulting library; they are loaded into shared library space if necessary when the program is run. Modules that reside in programs or other dynamic libraries are not copied into the resulting library; it is the programmer's responsibility to load these modules prior to loading the dynamic library. It is an error if the collection of implementations imports from more than one executable.

11. Dyn

`Dyn` is a short-cut that applies `Make-dynamic-library` to the result of `Bind`. Its definition is:

```
Dyn ~ LAMBDA pieces, ... IN
         Make-dynamic-library (Bind (pieces))
      END-LAMBDA
```

12. Bag

```
Bag: (name: text * bits: text * ... ) -> binding
```

`Bag` packages an initialized data object into a Vulcan object module and defines an interface that exports that object. Given a name `N` and a string, `bits`, of `len` characters, `Bag` creates and compiles the interface:

```
SAFE DEFINITION MODULE N;
VAR data: ARRAY [0..len] OF CHAR;
END N.
```

`Bag` also creates an object module that exports the interface `N` and contains the variable `data` initialized with the contents of `bits`.

`Bag` returns a binding with two elements; one for the compiled interface and another for the compiled object. In this example the returned binding would be:

```
{ N.d ~ <interface N>,
  N.o ~ <object N> }
```

13. Foreign

`Foreign` converts a Unix-format object file into a Vulcan object module. The resulting object appears to Vulcan the same as any other, except that there is no corresponding AST.

`Foreign`'s signature is:

```
Foreign: (object     : text *
          exports    : list *
          ... ) -> binding
```

where **object** is a Unix object file prepared as described below and **exports** is an interface or list of interfaces implemented by the object. The resultant binding is `{ X.o ~ <object> }`, where `X` is the name of the first interface in `exports`.

The supplied Unix object file must satisfy the following restrictions:

- The only external references must be to names defined in the imported interfaces. Programmers can use "`ld -r`" to link multiple Unix `.o`'s into an object suitable for use by `Foreign`.
- Only procedures, non-ref types, and constants can be exported by the exported interfaces. *Variables and ref types cannot be exported.*
- Procedures exported by the object must obey the Vulcan calling conventions; imported procedures are called using the Vulcan calling sequence.
- The programmer must supply an assembly-language stub that converts between calling sequences.

To prepare such a file, the programmer must first write an assembly language stub `Stub.o` that begins:

```
#include "/proj/topaz/friends/Vulcan.asdef"
VU_IMPORT (Wr)
VU_IMPORT (Rd)
BEGIN_PROCEDURE ( ...
```

This stub must convert between the Vulcan and Unix calling sequences. The Vulcan calling sequence is defined in "The Vulcan Runtime Architecture" available under `printdoc`.

To make a single `foreign.o` containing both `Stub.o` and other Unix `.o`'s:

```
ld -r -o foreign.o Stub.o ...other .o's...
```

It is this `foreign.o` that is passed to `Foreign`.

For each interface `I.d` imported by the foreign object, there should be a declaration:

```
VU_IMPORT (I)
```

This macro declares `I` as an external symbol

```
    .data
    .globl  I
```

and generates a linkage-record slot for I in the module's data segment.

For each imported interface I.d, Foreign will update the import map entry for I to consist of a one-element chain refering to its linkage-record slot. (Thus programmers don't need to worry about the order of imports.)

A procedure body can get the address of an item I.X in an imported interface record by doing:

```
movl    I,r1
moval   I.X(r1),r1
```

For each external symbol I.X referenced by the object file, if there is an imported interface I.d exporting a name X, Foreign will define I.X to be the offset of I.X in I.d's interface record.

Exported interfaces do not need to be explicitly declared. Foreign will add space for the exported interface records to the end of the data segment and update the export map to point to the records.

The object file exports a procedure E.P by using the following assembler macro at the entry point of the procedure:

```
BEGIN_PROCEDURE (E, P, frameSize, scopes)
```

This generates a partly filled-in procedure descriptor and defines the symbol E.P:

```
  .long   0  # next link, filled in by \verb|Foreign|
  .long   frameSize
  .long   0  # selfOffset, filled in by \verb|Foreign|
  .long   scopes
  .long   9f - E.P
  .globl  E.P
E.P:
```

The symbol scopes should reference the first of a linked list of scope descriptors (generated by the SCOPE macro) or be zero.

The programmer must conclude his procedure with END_PROCEDURE. This macro defines the label 9 that was referenced in the BEGIN_PROCEDURE macro.

For each procedure E.P in an exported interface, Foreign looks in the Unix object file for a global symbol E.P and updates the code segment's procedure map to point to the procedure descriptor referenced by E.P.

After processing exported and imported names, there should be no remaining undefined external symbols. If there are, Foreign generates an error.

14. ToBinding

ToBinding's signature is:

```
ToBinding: (value: any) -> binding
```

ToBinding converts the opaque values returned by the Vulcan bridge into Vesta bindings. The value passed to ToBinding must be an opaque value returned by the Vulcan bridge. Below is a table listing the opaque values and the contents of the returned binding.

```
interface (Foo.d)
   kind      : text = "Interface"
   uid       : text = the interfaces' Vesta UID
   name      : text = "Foo"
   source    : text = input source file
   timestamp : integer = creation timestamp
   imports   : list of imported interfaces (opaque)
   ast       : text = pickled ast of the interface

object (Foo.o)
   kind      : text = "Object"
   uid       : text = the object's Vesta UID
   name      : text = "Foo"
   source    : text = input source file
   timestamp : integer = creation timestamp
   imports   : list of imported interfaces (opaque)
   exports   : list of exported interfaces (opaque)
   ast       : text = pickled ast of the object
   code      : text = resulting object module

bound object
   kind      : text = "Bound-Object"
   uid       : text = the object's Vesta UID
   object    : opaque = input to Bind, an unbound object
   imports   : list of imported interfaces (opaque)

implementation (Foo.i)
   kind      : text = "Impl"
   uid       : text = the implementation's Vesta UID
   name      : text = "Foo.i"
   interface : opaque = implemented interface
   image     : opaque = image containing the implementation
   module    : integer = index into image's module map
   export    : integer = index into module's export map
```

```
program
   kind      : text = "Program"
   uid       : text = the program's Vesta UID
   contents  : list of bound objects (opaque)
   imports   : list of imported interfaces (opaque)
   bits      : text = the actual executable

load-and-go executable
   kind      : text = "Load-N-Go"
   uid       : text = the executables's Vesta UID
   contents  : list of bound objects (opaque)
   imports   : list of imported interfaces (opaque)
   bits      : text = the actual executable

shared library
   kind      : text = "Shared-Library"
   uid       : text = the library's Vesta UID
   contents  : list of bound objects (opaque)
   imports   : list of imported interfaces (opaque)
   bits      : text = the actual library

dynamic library:
   kind      : text = "Dynamic-Library"
   uid       : text = the library's Vesta UID
   contents  : list of bound objects (opaque)
   imports   : list of imported interfaces (opaque)
   bits      : text = the actual library

builtin function
   kind      : text = "Builtin"
   uid       : text = the function's Vesta UID
   name      : text = name of the function
   index     : integer = internal index of the function
```

15. Flume

Flume's signature is:

```
Flume: (source                     : interface *
        M2-clientBinding           : text *
        M2-serverBinding           : text *
        M2-clientAdjustsByteOrder  : text *
        M2-serverAdjustsByteOrder  : text *
        M2-marshalling             : list of text *
        M2-rpcProtocol             : integer ) -> binding
```

`source` is the compiled definition module to be *flumed.* The next four arguments translate into `flumeMain` switches as follows:

```
name                          mapping to 'flumeMain' switch
----                          -----------------------------
M2-clientBinding                "single" => nothing (*)
                                "multiple" => -xc
                                "pass-through" => -xc -l
                                "implicit" => "=ic"
M2-serverBinding                "single" => nothing (*)
                                "multiple" => -xc
                                "pass-through" => -xc -l
                                "implicit" => "=ic"
M2-clientAdjustsByteOrder       "never" => nothing (*)
                                "asNeeded" => -bc
                                "always" => -ac
M2-serverAdjustsByteOrder       "never" => nothing (*)
                                "asNeeded" => -bs
                                "always" => -as
```

The default settings are marked with `(*)`.

The `M2-marshalling` argument is a list of text values, which are passed as the explicit marshalling types and procedure names on the flumeMain command line. This list should have an even number of elements. By default, `M2-marshalling` is empty.

The `M2-rpcProtocol` argument specifies the wire protocol to use. Currently, only wire protocol 2 is supported.

The result binding contains names defined by `Flume`, as described in its manpage. If the definition file supplied as the `source` parameter has the name `Test` in its module header, then the binding will contain the following names:

```
    name                   present when
--------------             ------------
```

TestClient.def	always
TestClient.mod	always
TestRPC.mod	M2-clientBinding = "multiple" or "implicit"
TestServer.def	always
TestServer.mod	always
TestClientRef.def	Test contains opaque REFs
TestImpl.def	M2-serverBinding = "multiple"

16. Caulk

Caulk's signature is:

```
Caulk: (source          : interface *
        mode            : text
        M2-caulkTarget  : text) -> text
```

Caulk generates assembly language that converts the Vulcan calling sequence to either a Nub call, MM-style call or a C-style call.

The source argument specifies the interface to be caulked. The interface may only define procedures and non-REF types.

The mode argument specifies the type of calling sequence to generate:

```
 mode     calling sequence
 -----    ----------------------
"nub"    Nub call (chmk $NCN(module,proc))
"mm"     MM call  (callg _module_proc)
"cc"     CC call  (callg _proc)
```

The M2-caulkTarget arugment specifies the name of the targeted interface (i.e. module in the table above). If M2-caulkTarget is NIL, the name of the source interface is used. The default value of M2-caulkTarget is NIL.

The output of Caulk is an assembly lanague file that may be assembled (*until I can find the proper Vesta incantation*) by:

```
cc -E -I/proj/topaz/friends output.as | as -o output.o
```

The resulting .o file is designed to be linked with similar .o files so the result can be passed to the Foreign function.

Currently Caulk does not handle interfaces containing procedures with

- a fixed-length array parameter passed by value
- a return value that's an array
- a non-empty `RAISES` clause (except for nub procedures that raise `Base.Alerted`)

17. Rename

`Rename`'s signature is:

```
Rename: (value: any * name: text) -> binding
```

`Rename` *IS NOT IMPLEMENTED YET.*

`Rename` changes the name of an object or interface. `value` may be an object, interface, or a singleton binding that refers to an object or interface. The resulting binding uses the supplied `name` to refer to the new value.

For example,

```
Rename (Compile (MyWr.mod), "Wr")
```

will return the binding `{ Wr.o ~ <object> }` where `<object>` is a renamed copy of `MyWr.o`.

18. Analyze

`Analyze`'s signature is:

```
Analyze: (value: any) -> any
```

`Analyze` computes and prints some summary statistics about `value`. `Analyze` returns `value`, unchanged. `value` must be the result of some Vulcan bridge function.

19. Print

`Print`'s signature is:

```
Print: (value: any) -> any
```

`Print` is a simple tool for debugging models. `Print` returns `value`, unchanged. As a useful side effect, `Print` attempts to pretty-print its argument. The output of this printing is placed in a Vulcan window.

20. Errors

The bridge reflects all errors by raising the `Vesta.Error` exception with a descriptive text argument. In turn, the modeller reflects the error back to Vulcan and it displays the error to the user.