Writing Vesta Bridges

An Introduction to Integrating New Tools and Tool Flows into Vesta Builds

Ken Schalk / Writing Vesta Bridges
Tools under Vesta

- Tools run during a Vesta build have their context defined by an SDL program
  - The command line
  - The complete filesystem (see `chroot(2)`)
  - The environment variables
  - The standard input

- All these details are parameters to the SDL `_run_tool` primitive function
Tools under Vesta

- Setting up the context for a tool can be complicated
  - Tool executable file (for the right platform)
  - Run-time libraries, config files, etc. the tool needs
  - Placing the input files
- Vesta enforces precision (which is good)
- But users aren't interested in the details
Bridge = Abstraction

- A bridge is a collection of SDL functions that simplify running one more tools
  - “Bridges the gap” between the operation the user is interested in and the low-level details needed to carry it out
- Usually multiple bridges are gathered together into a build environment
  - A collection of different operations the user might need
Good Bridge Design

- Don't assume the filesystem is set up by the caller
  - Bridge function that runs a tool should add any files it needs (e.g. executable, run-time libraries, etc.)
  - For efficiency it's sometimes best to set up the filesystem once then run the tool multiple times

- Primary inputs should be parameters to bridge functions
  - Input files
  - Type of output/processing
Good Bridge Design

- Separate the SDL code from platform-specific files (executables, run-time libraries, etc.)
  - Possible to share bridge code across multiple platforms (e.g. x86 Linux, x86-64 Linux, Solaris)
  - Upgrade tool versions and change bridge code independently
  - Requires additional abstraction: bridge gets *specialized* to a particular platform with parameters by the build environment
Good Bridge Design

- Provide abstract *bridge options* for choices the user may want to make less often
  - Choices that affect multiple things
  - Anything likely to change between platforms
- Provide a way for the user to add arbitrary command-line switches to a tool
  - Handles situations the bridge writer didn't consider
  - Sufficient in many cases (e.g. “–DMACRO” for C/C++)
grep Bridge

- As an example, we'll write a bridge for a relatively simple tool: grep
- We'll use 3 Vesta packages for:
  1. The platform-specific files (the grep executable)
  2. The SDL bridge code
  3. An example of how to use the bridge
Getting the Executable

- Most modern operating systems use a *packaging system* to manage installed components
  - RedHat Linux uses the RedHat Package Manager (or RPM)
  - Debian Linux uses a different package manager
- Each installed *OS component package* is made up of some set of files
- *Package files* (.rpm/.deb) contain the files and all information needed to install a package
Getting the Executable

- To see the files in the *grep* package installed on Linux:
  - RedHat: `rpm -ql grep`
  - Debian: `dpkg -L grep`

- To get the files in the *grep* package into Vesta, we use the `pkg2vesta.pl` script
  - See: `/vesta/vestasys.org/vesta/extras/pkg2vesta`
Running `pkg2vesta.pl`

```
pkg2vesta.pl --from-installed \ 
  --package-root /vesta/example.com \ 
grep
```

- Creates the correct directory, package, branch
- Checks out the branch
- Fills the working copy with the files from the installed package plus SDL files and information about what was done

- If you have a package file (.rpm/.deb), don't use `--from-installed`
What pkg2vesta.pl Made

• root
  - Partial filesystem with all files from this package
• root.ves
  - Returns the filesystem
• build.ves
  - Returns the OS component ready for use
• README
  - What was imported, command-line options
If you Don't Have a `.rpm`/`.deb`

- You may not have a piece of software packaged for your OS:
  - It's being locally developed
  - It's only provided as source code for compilation
  - It was provided from a vendor in another form
- If you only have binaries, consider imitating the structure `pkg2vesta.pl` creates
- If you have source, consider compiling on demand under Vesta
Simple Bridge

```plaintext
{  // Search for pattern in file
grep(pattern: text, file: text): text
    {
        // Add the root for the tool and an empty working directory
        . += [ root = [.WD=[]] + ./build_root(<"grep">) ];

        // Build a command line
        cmd = <"grep", pattern>;

        // Run the tool
        r = _run_tool(.target_platform, cmd,
                       // Pass file as standard input
                       file,
                       // Capture standard output as a value
                       "value");

        return (if r == ERR || r/signal != 0 then ERR
                 else r/stdout);
    }

    // The bridge model returns this binding.
    return [ grep = [ grep ] ];
}
```
Simple Bridge Details

grep(pattern: text, file: text): text

- Defines a function named “grep”
- First argument “pattern” will be the pattern to search for
- Second argument “file” will be the text to search
- The function result will be the output of grep: the lines in “file” containing “pattern”
- Both arguments and the result are type text
- Like all SDL functions, this has a final implicit argument named “.”

Ken Schalk / Writing Vesta Bridges
Simple Bridge Details

. += [ root = [.WD=[]] + ./build_root(<"grep">) ];

- Augments the value of “.” with the binding overlay assignment operator (+=)

- Replaces “./root” with a new binding made by combining two bindings with the overlay operator (+)

[ .WD=[ ] ]

- An empty binding (i.e. directory) named “. WD”

- This is the default working directory when running a tool.
Simple Bridge Details

./build_root(<"grep">)

- Calls the function “./build_root” passing a list with a single element: the text string "grep"

- ./build_root is a convention used by build environments to make it easier to construct a root filesystem out of several OS component packages

- We just want the “grep” component we imported with pkg2vesta.pl, so that's all we ask for
Simple Bridge Details

cmd = <"grep", pattern>;

- Create a two element list holding:
  • The text string "grep"
  • The value of the “pattern” argument
- Store it in a variable named “cmd”
- This will be the command line we execute as a tool
Simple Bridge Details

```python
r = _run_tool('./target_platform', cmd, file, "value");
```

- This is where we actually run the tool
- The first parameter to `_run_tool` is the system type to run the tool on. We pass `./target_platform`.
- The second argument is the command line from our variable `cmd`
- The third argument is the standard input stream from the argument `file`
- The fourth argument is what to do with the standard output. We ask for it to be captured as a value.
Simple Bridge Details

• How is the filesystem passed?
  - In ./root
  - Like other functions, _run_tool takes "." as a final parameter, automatically from the calling context
  - We don't explicitly pass the filesystem, but _run_tool gets the value of "." implicitly, including the ./root we set earlier
Simple Bridge Details

• How are the environment variables passed?
  - In ./envVars
  - Just like the filesystem, environment variables are passed through “.”
  - We didn't set any here, but there might be some passed in as part of “.” from the caller of our grep function
Simple Bridge Details

return (if r == ERR || r/signal != 0 then ERR else r/stdout);

- After _run_tool finishes, we want to return the standard output of the tool
- First we check for a couple possible error cases indicating that the tool failed and return ERR if it did
- If all seems well, we return “r/stdout”
Simple Bridge Details

```c
return [ grep = [ grep ] ];
```

- The result of the bridge model is a binding meant to be made part of the “.” used by client models
- The binding contains the name “grep” with a binding value
- The `grep` sub-binding contains our `grep` function with its own name
  - Remember “[x]” is equivalent to “[x=x]”
- So users will get our function with ./grep/grep
Simple Bridge Usage Example

files
  // Sample text file to grep
  sample;
{
  // Find any lines containing the letter "a" in
  // sample. Put the result in a file named
  // "sample.out"
  return [
    sample.out = ./grep/grep("a", sample)
  ];
}
Putting The Pieces Together

• We now have several different pieces:
  - A package containing the `grep` binary we imported with `pkg2vesta.pl`
  - A package containing our bridge `build.ves`
  - A package containing our example usage `build.ves`

• To put them together, we need a platform-specific top-level model:
  `linux_i386.main.ves`
import
    self = build.ves;
from /vesta/vestasys.org/platforms/linux/redhat/i386 import
    std_env/9;
from /vesta/example.com/platforms/linux/redhat/i386/components import
    grep/"2.4.2-5"/1; // Our grep binary package
from /vesta/example.com/bridge_intro import
    grep_bridge/1; // Our grep bridge
{
    // Build the basic environment.
    .= std_env()/env_build([]);

    // Add the grep OS component package
    .+= [ components = grep() ];

    // Add the grep bridge
    .+= grep_bridge();

    return self();
}
Top-level Model Details

import

self = build.ves;

- The top-level model is in the same package as our example build.ves which calls ./grep/grep
- This imports the example build.ves, putting it in a variable named "self"
- We'll call it once we've set up everything we need for the example to work
Top-level Model Details

from /vesta/vestasys.org/platforms/linux/redhat/i386 import std_env/9;

- This gets the basic build environment for i386 Linux
  - It's based on RedHat 7.1, essentially a “lowest common denominator” environment

- It imports it into a variable named std_env
  - When an import doesn't contain “=”, the variable name is the first path component

- We'll use this for some basic things (.build_root among others) and augment it
Top-level Model Details

from /vesta-example.com/platforms/linux/redhat/i386/components import
grep/"2.4.2-5"/1; // Our grep binary package

- This gets the binary package we created with pkg2vesta.pl
- It imports into a variable named “grep”
- We need to quote the path component with the grep version number, because it contains “-”
  - Any path components containing characters other than letters, numbers, “.” and “_” must be quoted
  - Also, any path components matching reserved words must be quoted
Top-level Model Details

from /vesta/example.com/bridge_intro import grep_bridge/l; // Our grep bridge

- This gets the build.ves for our grep bridge
- It imports it into a variable named “grep_bridge”
  . = std_env() / env_build([]);

- This creates the basic build environment
  • Calls the std_env model
  • Looks up the name env_build in its result and calls it as a function
  • Puts the result in “.”
Top-level Model Details

. += [ components = grep() ];
- This adds our grep binary to the set of OS component packages stored in ./components
- After this, ./build_root will be able to build a filesystem including the grep OS component

. += grep_bridge();
- This adds our grep bridge to “.”

return self();
- Now that everything is set, call our example
Call Graph of Example

- **Example**
- **std_env**
- **grep Binary**
- **grep Bridge**
- **Primitive**

```
linux_i386.main.ves

std_env/9/build.ves

env_build

grep/2.4.2-5/0/build.ves

grep_bridge/1/build.ves

build.ves (usage example)

./grep/grep

./build_root

._run_tool

./components/grep/root
```
Call Graph of Example

```plaintext
linux_i386.main.ves

std_env/9/build.ves

env_build

grep/2.4.2-5/0/build.ves

. = std_env()/env_build([]);

. += [ components = grep() ];

grep_bridge/1/build.ves

. += grep_bridge();

build.ves (usage example)

./grep/grep

./build_root

./components/grep/root

_run_tool
```
Call Graph of Example

```
linux_i386.main.ves

return self();

return [
    sample.out = ./grep/grep("a", sample)
];

build.ves (usage example)

./grep/grep

./build_root

./components/grep/root

_run_tool
```
Call Graph of Example

```
linux_i386.main.ves

//build.ves (usage example)
./grep/grep

./build_root("grep")

root.ves created by pkg2vesta.pl

./components/grep/root

_run_tool (. /target_platform, cmd, file, "value");
```
Construction of Dot (.)

1. Basic . comes from std_env

```
. = std_env() / env_build([]);
```
Construction of Dot (.)

1. Basic . comes from `std_env`

```plaintext
. = std_env()/env_build([]);
```

2. We add our `grep` OS component

```plaintext
. += [ components = grep() ];
```
Construction of Dot (.)

1. Basic . comes from std_env
   
   . = std_env()/env_build([]);

2. We add our grep OS component

   . += [ components = grep() ];

3. We add our grep bridge

   . += grep_bridge();
Data Flow in `linux_i386.main.ves`

1. `std_env` returns `env_build`

```
[env_build=<function>]
```
Data Flow in *linux_i386.main.ves*

1. `std_env` returns `env_build`
2. `env_build` returns initial `dot`

```plaintext
linux_i386.main.ves -> std_env/9
[env_build=<function>]
[build_root=<function>, components=[gcc, libstdc++, ...], ...]
```
Data Flow in `linux_i386.main.ves`

1. `std_env` returns `env_build`
2. `env_build` returns initial dot
3. `grep` OS component model returns OS component (added to `.components`)
Data Flow in `linux_i386.main.ves`

1. `std_env` returns `env_build`
2. `env_build` returns initial `dot`
3. `grep` OS component model returns OS component (added to `.components`)
4. `grep` bridge returns bridge binding containing bridge function (added to `dot`)
Evaluating The Example

• When we try to evaluate our example, something seems to be wrong:

% vmake
Advancing to /vesta/example.com/bridge_intro/grep_example/checkout/1/1
Vesta evaluator, version release/12.pre13/5

0/hostname: grep a
0/Error: invoking _run_tool: /usr/sbin/tool_launcher: Execve failure, No such
  file or directory (errno = 2)
  Possible cause: perhaps tool path name is invalid or file system is
  incomplete?

One error was reported.
Vesta evaluation failed.

• Now we'll need to investigate
  – This is often part of writing a new bridge
Investigating The Problem

- We'll start by adding -fsdeps to the vmake command line:

```
% vmake -fsdeps
[...]
0/hostname: grep a
FS dependency: !/../root/.WD/grep
FS dependency: N/../root/bin/grep
FS dependency: !/../root/lib
FS dependency: !/../root/usr
0/Error: invoking _run_tool: [...]```

- This tells us that the tool is looking for some paths which don't exist:
  - /lib
  - /usr
Investigating The Problem

- Why don't /lib and /usr exist when the tool is running?
  - We specified its complete filesystem in /root before calling _run_tool
  - We only asked /build_root for the grep OS component
  - Perhaps the grep OS component with imported with pkg2vesta.pl doesn't have these directories?
Investigating The Problem

• Let's see what we imported:

% ls -lR /vesta/example.com/platforms/linux/redhat/i386/components/grep/2.4.2-5/1/root
/vesta/example.com/platforms/linux/redhat/i386/components/grep/2.4.2-5/1/root:
total 1
dr-xr-xr-x 1 ken root 512 May 27 14:43 bin

/vesta/example.com/platforms/linux/redhat/i386/components/grep/2.4.2-5/1/root/bin:
total 156
-r-xr-xr-x 1 ken root 49244 May 27 14:43 egrep
-r-xr-xr-x 1 ken root 49244 May 27 14:43 fgrep
-r-xr-xr-x 1 ken root 49244 May 27 14:43 grep

• Sure enough, no /lib or /usr
Investigating The Problem

- The `/lib` and `/usr` directories are probably not enough by themselves
  - The tool was probably looking for something inside one of those directories
  - Unfortunately, we don't know what
  - We could add empty `/usr` and `/lib` directories and run with `-fdeps` again to get more information

- Since it's looking for `/lib`, it's a good bet that it's a missing run-time library
Investigating The Problem

• Let's see what shared libraries our imported `grep` needs:

```
% cd /vesta/example.com/platforms/linux/redhat/i386/components/grep/2.4.2-5/1/root
% ldd bin/grep
  libc.so.6 => /lib/libc.so.6 (0x40025000)
  /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

• `grep` must be looking for the C run-time library (`libc.so`)
  
  - Most programs need this to run
  - We need to ask `.build_root` to include this for us when we call it
Fixing The Problem

• The name of the component with libc.so is “glibc”
  – This name is specific to the OS packaging system's naming convention, and may be different for other platforms

• To fix the problem, we'll change this:
  – ./build_root(<"grep">)

• To this:
  – ./build_root(<"grep", "glibc">)
Switches

• Suppose the caller wants to pass additional command-line flags to grep
  – \(-v\) to invert the match
  – \(-i\) for case-insensitive
  – \(-n\) to show line numbers

• Let's add code to allow users to add command-line switches to our grep invocation
Switches

- Define a place for users to supply switches as part of the bridge result:

```plaintext
// Optional command-line switches
switches = [];

// The bridge model returns this binding.
return [ grep = [ grep, switches ] ];
```

- This is similar to other standard bridges

- Users will add switches like:

```plaintext
. += [ grep/switches/invert = "-v" ];
```
Switches

• Inside the grep function, we'll incorporate the switches into the command line:

```c
// Build a command line
cmd = ("grep") + 
   
   ./generic/binding_values(.grep/switches) + 
   <pattern>);
```

• This uses a function from:

```
/vesta/vestasys.org/bridges/generics
```
Usage Example with Switches

- Let's use a switch in `build.ves`:

```
files
  // Sample text file to grep
  sample;
{
  // Ignore case when using grep
  . += [ grep/switches/nocase = "-i" ];

  // Find any lines containing the letter "a" or "A"
  // in sample. Put the result in a file named
  // "sample.out"
  return [ sample.out = ./grep/grep("a", sample) ];
}
```
Switches vs. Abstract Options

- We could instead create boolean options for these different grep capabilities:
  
  . +++ [ grep/options/nocase = TRUE ];
  . +++ [ grep/options/invert = TRUE ];

- We'd translate these abstract options into concrete switches in the bridge code

- This would be a good idea for complex options or options which use different command-line switches on different platforms
Multiple Files

• What if we have multiple input files?
  - The user could call `./grep/grep` multiple times
  - The bridge could support multiple files

• Let's add support for multiple input files
  - The input will be a binding rather than a single text value
  - We'll use the `par_map` primitive function to process the inputs
Handling Multiple Files

```c
grep(pattern: text, **pk***/inputs: NamedFiles): text
{
    // Add the root for the tool and an empty working directory
    . += [ root = [.WD=[]] + ./build_root("grep", "glibc") ];

    // Build a command line
    cmd = ("grep" +
           ./generic(binding_values(./grep/switches) +
           <pattern>));

    /**nocache**/
    grep_one(name, file)
    // Inner function that runs the tool for a single input file.
    {
        // Run the tool
        r = _run_tool(.target_platform, cmd,
                       // Pass file as standard input
                       file,
                       // Capture standard output as a value
                       "value");

        return (if r == ERR || r/signal != 0 then ERR
                 else [name = r/stdout]);
    }

    return _par_map(grep_one, inputs);
};
```
Details of Handling Multiple Files

grep(pattern: text, /**pk**/inputs: NamedFiles): text

- The second argument is marked with “/**pk**/” to tell the evaluator that the function's result will always depend on the complete value of this argument
  - This helps make caching more efficient
- The type “NamedFiles” means a binding whose values are all of type text
  - In other words, a directory that contains files but no subdirectories
  - See the vtypes(5) man page
Details of Handling Multiple Files

// Add the root for the tool and an empty working directory
. += [ root = [.WD=[]] + ./build_root(<"grep", "glibc">) ];

// Build a command line
cmd = (<"grep"> +
    ./generic/binding_values(.:/grep/switches) +
    <pattern>);

- We set up the filesystem and command line once, sharing it across all the individual grep runs

- Note that “cmd” gets captured from the definition context of the inner function, but the new value of “.” gets passed as a parameter (through _par_map)
Details of Handling Multiple Files

```c
/***nocache***/
grep_one(name, file)
  - We define an inner function which will run grep once for each input file
  - It must take two arguments (a name and a value) since we're going to use it with `_par_map` over a binding
  - We mark this function with "/***nocache***/" to suppress caching it

  • `_run_tool` is always cached, and this function doesn't do much besides call `_run_tool`, so there's no point in caching it
```
Usage Example with Multiple Files

- We changed the parameters to our function, so we need to update our `build.ves`:

```plaintext
files
    // Sample text files to grep
    inputs = [ sample1, sample2 ];
{
    // Find any lines containing the letter "a" in
    // our input files. (The bridge puts the results
    // in files with the same names as the inputs.)
    return .grep/grep("a", inputs);  
}
```
Finishing Touches: Generalization

- There are several things hard-coded in our bridge:
  - The command name ("grep")
  - The method for getting the root filesystem
  - The bridge name in the result ("grep")
- What if we wanted separate bridges for fgrep and egrep?
- Let's add bridge specialization parameters to remove these hard-coded parts
Bridge Parameters

• At the beginning of the bridge model, we'll add code which saves parameters from the value of "." when the bridge model is called

    // The command to invoke. (Optional parameter; defaults to "grep".)
    command = if .!command then ./command else "grep";

    // The root filesystem to use for this platform (which must include
    // the executable named by "command")
    root = ./root;

    // The name of this bridge. (Optional parameter; defaults to
    // "grep".)
    bridge_name = if .!bridge_name then ./bridge_name else "grep";
Bridge Parameters

- The value of “.” must be a binding containing the named parameters
- We have default values for `command` and `bridge_name`, using them if the caller didn't supply them
- The variables created here will be used below
  - Note that the definition of our function will capture these variables so they can be used when it is called
Bridge Parameters

• Inside our grep function, we'll use the `command` and `root` variables:

  ```
  // Add the root for the tool and an empty working directory
  . += [ root = [.WD=[]] + root() ];

  // Build a command line
  cmd = <command, pattern>;
  ```

• This assumes that:
  
  - `root` is a function which will return the root filesystem
  
  - `command` is a text value
Bridge Parameters

• At the end of the bridge model, we'll use `bridge_name` to change the name used in the binding returned:

  ```javascript
  // The bridge model returns this binding.
  return [ $bridge_name = [ grep, switches ] ];
  ```

• This assumes that `bridge_name` is a text value
New `linux_i386.main.ves`

```python
import
    self = build.ves;
from /vesta/vestasys.org/platforms/linux/redhat/i386 import
    std_env/9;
from /vesta/example.com/platforms/linux/redhat/i386/components import
    grep/"2.4.2-5"/1; // Our grep binary package
from /vesta/example.com/bridge_intro import
    grep_bridge/1; // Our grep bridge
{
    // Build the basic environment.
    . = std_env()/env_build([]);

    // Add the grep OS component package
    . += [ components = grep() ];

    // Add the grep bridge
    bridge_args = [ command = "grep", bridge_name = "grep",
                    root = ./build_root_delayed("<grep", "glibc") ];
    . += grep_bridge(bridge_args);

    return self();
}
```
Top-level Model Changes

```plaintext
bridge_args = [ command = "grep", bridge_name = "grep",
                root = ./build_root_delayed("grep", "glibc") ];
```

- This sets up the bridge specialization arguments
- `.build_root_delayed` is like `.build_root`, but it returns a function which will build the root filesystem later

```plaintext
++= grep_bridge(bridge_args)
```

- This passes the arguments to the bridge model as “.”
  
  - Remember: imported models have one implicit argument which is “.”
Learning More

• Examples from this presentation can be found in:
  – /vesta/vestasys.org/examples/bridge_intro
  – See the README file for some suggested exercises

• The lex bridge dissection in the SDL reference is another document which can help you learn about bridge writing:
  http://www.vestasys.org/doc/sdl-ref/bridge-dissection.html
Learning More

• The full documentation of the _run_tool primitive function describes capabilities not covered here:


• Read the code of std_env and other bridges
  – There's no special magic: it's all just a library of SDL code for calling _run_tool, and now you've seen how it works