Virtual Laboratory Abstract Machine Model for Module Writers

Berry van Halderen

DRAFT on July 5, 2000

Contents

This is a document which is still under discussion, incomplete and under *construction.*

1 Introduction

Within the virtual laboratory project, a non-expert user can compose an experiment by selecting different modules and combining them to perform a certain task. Sometimes, the modules selected by the user may actually be constructs of a set of modules to hide lower-level details or operational constraints. However when a user composes an experiment this will result is a graph of connected basic modules which interact to perform the task defined. This graph can be executed on the execution platform of the virtual laboratory. The execution platform is the implementation of the virtual laboratory abstract machine model.

Because the modules may perform large amount of computational work, or because resources the modules are bound to may be located at different computing hosts, the execution platform of the virtual laboratory is distributed.

The basic modules in the virtual laboratory are independent programs which have to conform to the interface defined in the abstract machine model. Modules are written by module writers and can be added to the virtual laboratory through an administrative process.

Module writers should use the interface offered by the abstract machine to interact with other modules, the environment and the execution platform itself. This Application Programmer's Interface (API) is described in this document and is mandatory for a correct operation of the modules within the virtual laboratory.

1.1 Modules in the abstract machine model

Within an experiment setup a certain function may be needed multiple type. Multiple instances of the same module may therefor exist and we should thus distinguish between a module definition and a module instance. A module definition consist of a piece of program code and some static data structures. An instance of this module executes this program code.

Within the virtual laboratory model, modules are continuous active processes. This means that when an instance is created it is called only once, and the program can or should enter a while loop itself to continue processing on data that is coming in. This is more flexible than a call-return mechanism, where execution needs to be stateless, or state should explicitly be maintained. In that case the programmer should need to encode where to continue processing the program code when called a next time.

Depending on the implementation, multiple module may be concurrently executing within a single process using separate threads of controls for each module instance.

Modules can interact by passing information over streams, which connect one module to another. These streams are created by the runtime system and connected to the proper modules as defined in the setup of all modules. Modules themselves interact transparently with each other. This means that a module does not address the peer module directly. Instead they simply place data into the streams and retrieve data out of them, without knowing which type of module is at the other end of the stream.

For this process to work, modules need to define a number of ports to initiate this communication over. When an instance of a module is created, the module should define it's ports, whether they are input or output ports and the datatype that is communicated over it. The execution platform will then take care of connecting the port of the module to the port of the module to which it is connected.

Communication over streams is pure first-in first-out based, where only one element of data is being placed into the head of the stream at a time, and only one element can be retrieved at a time. All elements of data transmitted over the same port must be of the same time. Ports are therefor typed, and the types associated with two connecting ports must match.

2 Standard definitions

2.1 Naming conventions

In the C API all module methods, variables and constants have the prefix `vlab_` or `VLAB_`. The current exception for this normal C convention are structure and union names. These names are language level constructions, which will not conflict with existing names on linker level. *It may be well advised to change this and also prefix structure and union names.* Constant values and macros are capitalized, while variables, type definitions and functions are in lower case. Type definitions (C typedef constructs) have the postfix `_t`. The library reserves names starting with `vlab__` and `VLAB__` for internal use. Modules should not try to make use of internal methods or use a specific not-described behaviour of the interface.

2.2 Errors

Most methods return an integer value. This integer value is denotes the possible error that occurred. A negative return value indicates an error, while zero indicates a successful operation. A positive value currently is undefined, to check for an error, one has to check whether the result code is negative.

The API provides definitions to check for a specific error. The special return code `VLAB_OK` indicates a successful operation is guaranteed to be `0`. Error codes are global variables or definitions starting with the prefix `VLAB_E`. Current error return codes are:

`VLAB_ENOTIMPL`	function not implemented
`VLAB_ENOTAVAIL`	resource not available
`VLAB_ENOTDEF`	named entity not defined
`VLAB_EBADTYPE`	bad data-type as parameter

Possible error return values differ per method.

The method `vlab_strerror` can be used to get a human interpretable description of an error that has occurred.

The return string should be used for human readable output only and may be dynamic. It should be called directly after an error is detected and before any other vlab procedure is called. The return string may be volatile and a second call to `vlab_strerror` may invalidate any previous returned string.

The prototype for this function is:

```
extern char *vlab_strerror(int code);
```

2.3 Type definitions

The API interface provides a number of type definition to refer to different constructs in the Virtual Laboratory model. Such as a module, port, type, etcetera. These type definitions represent abstract handles to such an entity in the abstract machine model and are opaque, i.e. they do not reveal internal structure or implementation.

These handles are passed to the modules or returned by some of the methods in the API interface. Some methods require the module to pass one or multiple handles to identify a specific object in the abstract machine.

The scope of all handles is at least limited to a single instance of a single module. There is no way in which to share or pass handles to other instances. The scope of handles may further be limited by other constructs in the interface.

`vlab_module_t`

Specifies an opaque handle of a module. A module is given this handle by the runtime system when initialized or when the module is run, and should be used in certain calls back into the runtime system. This provides a method for the runtime system to know which module performs a request and distinguish between them.

`vlab_buffer_t`

A handle to a buffer space, which is used in reading and writing data. Data is read into a buffer space (through the use of a read-call). When the buffer space is no longer needed it's released to make it's resources available. To write data, a buffer space must be allocated, filled with data, and then written. Reading and writing in the Virtual Laboratory take place on buffer spaces, not on plain memory.

`vlab_port_t`

A handle to a port, such as defined in the abstract machine model. A port is either an input-only port or output-only port. Input ports may be used to read data from, while output ports may be used to write data.

`vlab_parameter_t`

Specifies an opaque handle to a parameter. Parameters are variables which can be set for the controlling environment (user front-end), while they can only be read on by modules.

`vlab_state_t`

Specifies an opaque handle to a state of a module instance. These are variables which can be set by a module instance and can be read back by the controlling environment (user front-end).

`vlab_datatype_t`

Specifies a handle to a data-type. A data element that is written to a port or read from a port must be of one data-type. This data-type is represented by this handle.

`vlab_datafield_t`

Represents a specific field within a certain data-type, such as a field within a structure. These are needed in more complex data-types with variable with fields.

3 Module definitions

To load a module into the virtual laboratory platform, a certain static structure must be defined in which certain entry-points are specified.
The life-cycle of a module definition as seen from the execution platform starts from this point.

In order to allow future expansions of the specification of a module, the first thing specified is the version number of this specification. Version 0 is reserved for internal use.
Version 1 of this specification structure is defined as:

```
struct vlab_specification {
  version-number,
  "name",
  initialization-function,
  run-function
};
```

No assumption on the internal naming of the structure should be made and the structure should be static and have a global linking-scope.

The execution life-cycle of a module is as follows:

- Specification structure is retrieved from the module by the runtime platform.
- A module is instantiated by calling the *initialization-function*. At this stage within this function a module
  - should: declare the data-types used by ports
  - must: declare the module instance ports
  - must: declare all parameters and states
  - can: read arguments

  This should be done in this sequence. The number and type of ports may not be dependent on the arguments accessed. Nor may the parameter or states defined by a module. All other operations, such as interaction with other modules in the system is not allowed at this stage.
- Instances of modules are started by calling the *run-function*.
  - itemize
  - can: interact with other modules by reading and writing data through the use of buffers.

  - can: update parameters, to the most recent state known at that time by the runtime-system (which may not be the most recent state as seen by the user).
  - can: update a state, which change will propagate back to the controlling environment.

  All other operations, such as declaring ports, data-types, parameters or states are prohibited.

- When a module instance has exhausted all function, it may exit to the calling environment. Allocated resources (buffers, allocated memory, a.o.) in both the initialization-function and run-function should be freed or released.

3.1 Buffer usage

Therefor a buffer also has a life-cycle, a buffer is either inward bound or outward bounds. Inward bound buffers are used to process read data. Outward bound buffers are objects in which data can be placed to be written.

An inward bound buffer is valid from the point when the handle is set (returned) by a call to the `vlab_read` method, and remains availably until explicitly released by a call to `vlab_release`.

There may be multiple inward and multiple outward buffers in use by a module at any one time, through it is advisable to have a nested usage of buffer.. A nested structure of allocation/deallocation means that if you first allocate a buffer a (either through `vlab_prepare` or `vlab_read`), and then a buffer b, then you should first deallocate b (through the corresponding `vlab_write` or `vlab_release`) and then a.

Can this be required, at least on a per-port basis?

4 Type definitions

Modules interact by passing data. This data must be of a certain type, which is related to the type declared by the ports. In order for the execution platform to successfully pass data from one module to another, it must be made aware of the structure of the data. Declaring this is done through a number of methods in the runtime system, which allow the module developer to build such a type definition.

This does not yet handle the definition of types in a separate library.

Once a type has been committed, no modifications can be made to it (i.e. no fields can be added to composed types). A committed type must be complete, this means that when the type contains a variable size array, the field which specifies the size must be a field within the type.
Only committed types may later be used in the communication between modules.

The typing system allows you to build complex data-types such as structures and arrays out of basic elements.

The following methods construct a basic type or atomic type:

```
extern int vlab_type_char (vlab_datatype_t *result);
extern int vlab_type_unsigned_char (vlab_datatype_t *result);
extern int vlab_type_byte (vlab_datatype_t *result);
extern int vlab_type_short (vlab_datatype_t *result);
extern int vlab_type_unsigned_short (vlab_datatype_t *result);
extern int vlab_type_int (vlab_datatype_t *result);
extern int vlab_type_unsigned (vlab_datatype_t *result);
extern int vlab_type_long (vlab_datatype_t *result);
extern int vlab_type_unsigned_long (vlab_datatype_t *result);
extern int vlab_type_float (vlab_datatype_t *result);
extern int vlab_type_double (vlab_datatype_t *result);
extern int vlab_type_long_double (vlab_datatype_t *result);
extern int vlab_type_long_long_int (vlab_datatype_t *result);
```

When called, the reference to a type-handle as given in the first parameter to the C function is set to an atomic type definition corresponding to the type requested. Multiple calls to the same constructor function may return different pointers representing the same data type. When two atomic datatypes are needed to construct a type, the programmer must call the same constructor function multiple times, once for each instance of the atomic type needed.
For example, if the module wants to define a structure of two integers, then the module should call the `vlab_type_int` twice and add both types into the structure definition. It is not allowed to call `vlab_type_int` once and add the type twice to the structure definition.

More complex data structures are build using the following composed datatypes:

```
extern int vlab_type_struct (vlab_datatype_t *result);
extern int vlab_type_array (vlab_datatype_t *result,
 int size);
extern int vlab_type_vararray (vlab_datatype_t *result,
 vlab_datafield_t size);
extern int vlab_type_add (vlab_datafield_t *result,
 vlab_datatype_t composed,
 vlab_datatype_t subtype);
extern int vlab_type_commit (char *name, vlab_datatype_t type);
extern int vlab_type_named (vlab_datatype_t *t,
 char *name);
```

The `vlab_type_struct` method initializes a C structure type definition. Each subsequent field of the structure must be added to calls to `vlab_type_add`. All fields of a C struct definition must be added in the same order as they appear in the C structure. Thus if we have a C structure:

```
struct t {
  int a;
  float b;
  struct {
    long d;
  } c;
```

Then the type definition would look like:

```
  vlab_type_int(&ta);
  vlab_type_float(&tb);
  vlab_type_long(&td);
  vlab_type_struct(&tc);
  vlab_type_add(NULL, tc, td);
  vlab_type_struct(&t);
  vlab_type_add(&f, t, ta);
  vlab_type_add(NULL, t, tb);
  vlab_type_add(NULL, t, tc);
```

All fields of the structure must be added directly after a call to `vlab_type_struct`. Any composed type such as build using `vlab_type_struct` may not be referred to in other type constructing calls before all fields are added.

The `vlab_type_add` call is used for multiple composed type building methods and should be given the following arguments:

1. A pointer to handle which will represent the field of the structure (or other data-type) in methods which query or set an item in a data-structure. This is the method of the virtual lab typing system to name a specific field in the data structure, much like in C you would be able to access the field `a` in the data structure `t` by its name "a". In the typing system this field is identified using the handle `f` in the example above.
   This argument may be `NULL` if the field is not needed.
2. A handle for the composed type to which this field should be added. The type must be a composed type and may allow for additional restraints.
3. A handle to the type of the field which is to be added. The typing system will take care of necessary padding due to alignment of different types.

An fixed-size array of identical typed elements is created using the method `vlab_type_array`. The second argument indicates the size of the array as an element count. A variable-sized array can be build using `vlab_type_vararray`.

The virtual laboratory runtime system does however need to know how large this array is. To solve this, any data-type that is committed must be completely self-defined. This means that when such a committed data-type contains a variable-sized data-type, then the size of this data-type must be given by one of the fields which is also contained within the same committed data type. Naturally this size cannot be specified within the variable-sized array.

A proper variable-sized array definition may look something like:

```
struct t {
  int a;
  float *b; /* array b has a elements */
};
  vlab_type_struct(&t);
  vlab_type_int(&ta);
  vlab_type_add(&f, t, ta);
  vlab_type_float(&tf);
  vlab_type_vararray(&tb, f);
  vlab_type_add(NULL, t, tb);
```

Finally the `vlab_type_commit` commits a data-type and makes it available for further use. Any committed datatype has an associated name with it as defined by the first argument to `vlab_type_commit`. This name may be used to retrieve a committed data-type using the `vlab_type_named` which returns a handle to the, in the second argument, named data-type in the handle pointed to in the first argument.

Auxiliary typing interfaces

The following auxiliary methods are furthermore available:

```
extern vlab_datatype_t vlab_gettype(char *type);
extern void vlab_setsize(vlab_datafield_t field, int value);
```

The `vlab_gettype` is currently synonymous for the `vlab_type_named` call. The `vlab_type_named` is intended to retrieve a type when building a new type, while the `vlab_gettype` is intended to retrieve a committed type in order to pass it to one of the methods where a type is associated with an item (stream, parameter, ...) such as the methods `vlab_input`, `vlab_output`, `vlab_declareParameter`, etcetera.
Future specification on this subject may arise.

The `vlab_setsize` method is discussed in more detail later. It sets the value of a field which has previously be defined using a call to `vlab_type_add` to the specified value, before the actual data type has been allocated. This allow for defining the size of variable sized arrays where the typing system will allocate the bufferspace. It is currently only relevant for fields of type `vlab_type_vararray`.

The method name `vlab_setsize` may be replaced by a more generic type when unions (a.k.a varrecords) are added.

5 Interaction between modules

Modules receive data from other modules and pass data to other modules through data streams. A data steam is a connection of an output port of one module to the input port of another module. These ports are defined by the module in the initialization of the module, and are of a static nature. The number and types of the port (both whether it is input or output and the type of data send over it) must be the same for every instantiation of the module within any configuration. All ports must be defined in the initialization of an instance of a module and no communication can take place in the initialization phase. When a module is running, no ports can be added or modified.

Inward bound ports are defined with `vlab_input` and outward bound ports using `vlab_output`. These two functions can only be called during the initialization of a module instance. The definition of these two methods is:

```
extern int vlab_input(vlab_module_t, char*, vlab_datatype_t, vlab_port_t *);
extern int vlab_output(vlab_module_t, char*, vlab_datatype_t, vlab_port_t *);
```

When called, they require the module initialization to pass the `vlab_module_t` handle, as passed to the initialization function of the module by the runtime system. The second parameter to these modules names the port, this is a null terminated C string which is used to let the environment tell apart one port from another. The `vlab_datatype_t` handle must be a committed datatype which represents the data-type that will be send or received over this port. The final argument is a pointer to an uninitialized `vlab_port_t` handle. The `vlab_input` or `vlab_output` methods will initialize this handle to represent the port defined by the given name.

Modules should use the port handle set by these functions in future calls to the following communication calls.

Communication over ports is defined using read and write like operations. Single elements of data are read or written over the port from or into the data-stream. Any of the read and write operations (including the prepare method) can not be performed when the module instance is initialized. All elements written or read are of the same type and of the type as defined when the port was declared.

5.1 Reading data

To read an element of data the `vlab_read` method is used. This inputs one element from the head of the data-stream. The read operations have however a somewhat different form. Instead that the destination where the data element should be placed is passed to the method, the method returns the place where the data is stored. This buffer space memory is represented by a handle (`vlab_buffer_t`). When a module is done with the data read, the bufferspace must be released. To identify the bufferspace memory a separate handle is also returned by the `vlab_read` call. This handle should be passed to the `vlab_release` method, after which the bufferspace should no longer be accessed.

The prototype of the read and release call is:

```
extern void *vlab_read(vlab_port_t, vlab_buffer_t *, long);
extern int vlab_release(vlab_buffer_t);
```

The `vlab_read` method reads an element of data from the stream connected to the specified port (`vlab_port_t`) with some options specified in the bit field of the `long` type. It returns a handle to the bufferspace pointed to in the second argument and returns a pointer to the actual data in the buffer as the return value. A return value of `NULL` identifies an error.
The `vlab_release` method takes a previous returned `vlab_buffer_t` handle and recycles the bufferspace.

5.2 Writing data

Writing data is much like reverse of the reading process. To write data into a stream a port is used to identify the destination. Also the instead of directly writing a piece of data from user space memory, a bufferspace must be used. To write data therefor a module makes use of the `vlab_write` method which takes only one argument, the bufferspace that should be written over the port. This bufferspace should earlier have been requested using a call to `vlab_prepare`. `vlab_prepare` allocates and prepares a bufferspace to write over the specified port in the first argument to this method. Some options may be provided in the bit field of the last argument and a handle is returned in the handle pointed to in the second.

```
extern void *vlab_prepare(vlab_port_t, vlab_buffer_t *, long);
extern int vlab_write(vlab_buffer_t);
```

After the call to `vlab_write` the bufferspace is no longer valid.

5.3 Variable length data in communication

The virtual laboratory typing system offers the possibility to express dynamically sized data in communication. This relatively unique method gives the flexibility of being able to define a broad amount of types. However in combination with the two phase communication scheme where bufferspace must be allocated using a call to `vlab_prepare` this may initially look complicated.

The problem is that when a bufferspace is requested through `vlab_prepare` with a data-type which contains a variable-sized array, the size of this array is unknown and that thus the typing system cannot set-up this bufferspace. This is confronted by letting the module writer first specify the size of these arrays directly before the `vlab_prepare` call. The sizes of the variable-sized array are normally given by fields somewhere else in the data-structure. Using the handle returned by `vlab_type_add` we can indicate the typing system that when initializing the data-structure this field, which represents the size of the variable-sized array, should be initialized at a certain value.

This is done using a call to the `vlab_setsize` method. When passed the handle to a field and a value, it will make sure that this field is initialized at the specified value. In this way the field which gives the size of the variable-sized array is initialized just in time before initializing the array itself.

In an earlier example we used the variable-sized array type definition:

```
struct t {
  int a;
  float *b; /* array b has a elements */
};
  vlab_type_struct(&t);
  vlab_type_int(&ta);
  vlab_type_add(&f, t, ta);
  vlab_type_float(&tf);
  vlab_type_vararray(&tb, f);
  vlab_type_add(NULL, t, tb);
```

Preparing a bufferspace and writing this would be expressed as follows:

```
  vlab_setsize(f, 100);
  struct t *v = vlab_prepare(port, &buffer, 0);
  /* v->a is defined as 100 */
  for(i=0; i<v->a; i++)
    v->f[i] = 0.5F;
  vlab_write(buffer);
```

Where `port` should have been defined as an output port of type `t`.

6 Interaction of a module with it's controlling environment

Modules may interact in an asynchronous (and relatively slow) method with its controlling environment, such as a Graphical User Interface (GUI). This interaction has two directions:

1. The user may, at runtime, change certain values which are used by the module. These variables are called *parameters* in the virtual laboratory system.
   The controlling environment may only set the value of a parameter, while the module instance may only retrieve the value of a parameter (and not set it).
2. A module instance may pass a value to the user environment indicating a certain *state* of the module during runtime.
   A state may only by set by a module instance and can only be queried by the controlling environment.

Parameters and states can be considered environment variables for modules and deliver a completely orthogonal complementary service.
In order for a consistent view, it is not possible to build a environment variable which represents both a parameter and state in the above definition. This holds for both the model and API to the module and for any possible front-end.[1]

Both parameters and states are only valid when all modules have been started (i.e. their run-method is invoked. For instantiating a module, there is another type of environment variable available;

4. An *argument* is a value which is passed to a module from the controlling environment. Arguments may only be queried by the module instance during it's instantiating method and can be considered static values.

An argument value can be queries using the following method:

```
extern void *vlab_getArgument(vlab_module_t, char *, vlab_datatype_t);
```

The second argument to this method specifies the name of the argument, the scope of an argument (name) is that of within a module so the module handle has to be given in the first argument. The third argument specifies a committed

[1] A GUI may represent the value to the user as one, but even this may be not advisable!

data-type which specifies the type of the data. The `vlab_getArgument` call returns a pointer to the argument data (i.e. the value) or `NULL` if the argument is not set or of a different type.

6.1 Parameters and states

Both states and parameters have to be declared during the instantiation of a module using a declare call. States and parameters are named with a scope limiting them to a module instance. The first argument to a declare method is the handle to the module and the second the name of the environment variable. The third argument specifies the previously committed data-type which represents the data type of the environment variable. Parameters and states are also stored in bufferspaces allocated by the typing system, however once declared they remain available during the entire life-time of a module instance. A handle to this bufferspace is set to the reference pointed to by the fourth argument to the declare call which is either a `vlab_parameter_t` or `vlab_state_t` depending on the type of environment variable being declared. A pointer to the actual storage of this data is passed back as return value.

```
extern void *vlab_declareState(vlab_module_t, char *, vlab_datatype_t,
       vlab_state_t *);
extern void vlab_updateState(vlab_state_t);
extern void *vlab_declareParameter(vlab_module_t, char *, vlab_datatype_t,
   vlab_parameter_t *);
extern void vlab_updateParameter(vlab_parameter_t);
```

A call to an update method will update the parameter or state. For a parameter this means that the most recent change as known by the runtime system of this parameter by the controlling environment is copied into the bufferspace as specified and returned by the declare method. If the control environment has changed the parameter multiple times before a call to `vlab_updateParameter` is performed then only the last change is seen by the module.
In between calls to `vlab_updateParameter` the parameter value in the bufferspace remains static. Only a call to this method will update the parameter.

Because module instances and controlling environment operate asynchronously, changes made by the environment may not directly be visible to modules. Due to communication, which must be considered slow for this application, changes may take time to be received by a module. Parameters are therefor updated to the most recent value known and not synchronized to the controlling environment.

Updating states works similar. The module can change the value of a state as it pleases but the change is not seen by the controlling environment until it performs a call to `vlab_updateState`. State values are also not synchronized with the controlling environment, but changes propagate relatively slowly to the controlling environment. The interface of the controlling environment will also return the most recent value known, such that previous state changes may be skipped.

Both `vlab_updateState` and `vlab_updateParameter` can be considered as low-overhead calls as the runtime system will try to minimize communication at the expense of an a-synchronize update scheme between the modules and controlling environment.

A Compiling a module

The `vlabc` utility is available for compiling a module into a shared library, such that it can be loaded by the virtual laboratory execution platform.

```
User Commands                                           VLAB(1)


NAME
     vlabc - vlabc, compile virtual-lab modules

SYNOPSIS
     vlabc [ -g ] -o outfile files ...

DESCRIPTION
     vlabc is a tool for building modules  for  the  Virtual  Lab
     system.   This  is  achieved  by  building  a dynamic shared
     object from one or more source or object files.

     The argument files can be any  C  source  file  (.c)  or  an
     object  file  (.o).  The vlabc tool automatically recognizes
     these extensions and automatically uses the C  source  files
     for  compilation while it just uses the object files for the
     linking phase. But when  using  such  pre-compiled  objects,
     make sure they are compiled for position independend code.

OPTIONS
     -n outfile

     Instructs the command to output the  module  dynamic  object
     into  the  file  outfile This option must be present and you
     should confirm to the platform naming scheme.  Normally  the
     outfile on many platforms ends with the extension .so

     -g

     When compiling C source files, pass the option -g to  the  C
     compiler

FILES
     none.

SEE ALSO
     vlab(1), cc(1).

DIAGNOSTICS
     Largely untested, please report problems, inconsistencies or
     other unsuspected behaviour.

BUGS
     This script cannot handle filenames containing  whitespaces,
     and  other non-alphanumberic characters which normally cause
     problems in shell scripts.
```

B Experimental platform

An experimental platform for testing modules and running experiments has been setup up. The platform has been prototyped using the scripting language and toolkit Tk/Tcl. Basic knowledge of Tcl is required to work with this tool. Knowledge of Tk is only needed when creating a graphical interface to the experiment.

Tcl is an extensible scripting language which means that it has the possibility to add new commands into the language. This allows you to quickly add a command interface to one owns application which is powerful and allows for programming. This is the way in which it has been used for the prototype. The prototype tool offers a Tcl command line interface, in which the user has the availability of an additional command by which modules can be loaded, configured and a setup can be executed.

The program to run for the graphical environment is `vlabinterface`, and the prototype tool for non-Tk support is `vlabplatform`. Both programs offer a Tcl prompt where the Tcl command `vlab` has been added.

On the command line prompt the following vlab commands can be given:

> `vlab load` *full-path-to-library specification-structure-name*
> `vlab create` *configuration-name*

The load operation loads a module which has been compiled into a shared library (using vlabc). The full pathname has to be given because the tool currently does not query the library from the current directory. Furthermore the name of the `vlab_specification` structure which defines the entry-points into the module has to be given.

The create operation gives the start point to be able to define a executable setup of modules (configuration). This command will initialize some data structures and add a new command with the same name as the configuration-name to the Tcl interpreter. The configuration-name therefor has to comply with the normal naming schemes of Tcl, and you should normally restrict to alphanumeric characters starting with an alphabetic character.

After the create operation has been given the following commands are available:

> *configuration-name* `add` *instance-name module-name*
> *configuration-name* `connect` *instance-name*$_1$ *port-name*$_1$ *instance-name*$_2$ *port-name*$_2$
> *configuration-name* `start`

The add operation creates an instance of the module named *module-name* and names this instance *instance-name*. The module name is the name as specified in the specification structure `vlab_specification` in the module source. The instance name is only relevant for this interface tool and bears no meaning for the executing modules (since they should not refer to each other directly).

The connection operation allows you to connect two modules together. In a port-based system as the virtual laboratory abstract machine model, there may be multiple ports from and to which to connect the stream to. Therefore for both the producer (writing into the stream) and consumer (reading from the stream) modules the port has to be given. The connection operation connects the producer module 1 to consumer module 2. The instance-name of both modules is the instance name as previously given to a add operation, while the port-name arguments are the ports as defined by the modules using the `vlab_input` and `vlab_output` methods.

Finally the start operation can be given after all modules are fully defined, arguments have been set and the modules have been given to run the configuration. The start operation is guaranteed to return only when the run-method of all module instances has been called.

When a module instance has been defined using the add operation, a Tcl command is added with the instance-name as command name by which the module instance can be controlled. The following operations are now possible on this command:

instance-name `set` *argument-name value*
instance-name `put` *parameter-name value*
instance-name `get` *state-name count*

The set operation sets an argument available in the initialization phase of an instance of a module. The argument-name should be the same as defined by the module in the `vlab_getArgument`. Set operations are only allowed before a configuration is executed (i.e. the start operation is been given). The put and get operations allow you to set and query a parameter or state, respectively. The parameter-name should be the same as defined by the module in a `vlab_declareParameter` and the state-name should be the same as defined in a `vlab_declareState` method call. Both put and get operations are only allowed after a configuration is given the start command.

The put operation sets the named parameter in the instance specified by the command name to the value given as final argument to the command. The get operation retrieves a named state. The current interface allows for an array of integers to be returned, however because the interface has no knowledge of the possible size of this array this size must be passed as an argument (count) to the get operation. The get operation returns a list of values which can be retrieved using the normal Tcl commands such as lindex.

Currently only integer values are allowed for the set and put operations, and the result of the get operation is restricted to an integer or an array/list of integers. The syntax of the get operation is likely to change in future implementations, as may be set and put operations.

One should first define all module instances, then set arguments to the modules and only after all these have been performed connect modules to each other. When the configuration has been started, no more instances should be defined, arguments be set or connections be made.

Usage

The tools for the Virtual Laboratory prototype may be found in the directory

```
nfs://carol.wins.uva.nl/home/berry/projects/vlab
```

in which there are subdirectories named “bin”, “man”, and “include” for the executable programs, manual pages and include files.

C Example module

The following source code specifies a module which reads a sequence of integer values and builds a histogram of it. This histogram is periodically send out. The interval after which to send out the histogram is controlled using a parameter and the number of elements processed is available as a state environment variable.

```
#include <stdlib.h>
#include "vlab.h"

struct mod_hist_info {
  long range;
  long *aggregates;
  vlab_port_t in;
  vlab_port_t out;
  int *count;
  int *threshold;
  vlab_state_t countref;
  vlab_parameter_t thresholdref;
};

static void *
initialize(vlab_module_t handle)
{
  int i;
  vlab_datatype_t t1, t2;
  vlab_datafield_t f1;
  struct mod_hist_info *info;
  info = malloc(sizeof(struct mod_hist_info));

  vlab_type_long(&t1);
  vlab_type_commit("long", t1);

  info->range = *(long *)vlab_getArgument(handle,"range",vlab_gettype("long"));

  vlab_type_long(&t1);
  vlab_type_array(&t2, info->range);
  vlab_type_add(&f1, t2, t1);
  vlab_type_commit("longarray", t2);

  vlab_input(handle, "in",  vlab_gettype("long"), &info->in);
  vlab_output(handle, "out", vlab_gettype("longarray"), &info->out);

  info->threshold = vlab_declareParameter(handle, "threshold",
  vlab_gettype("long"),
  &info->thresholdref);
  *(info->threshold) = 1;
  info->count = vlab_declareState(handle, "count",
  vlab_gettype("long"),
  &info->countref);

  info->aggregates = malloc(sizeof(long) * info->range);
  return info;
}

static void
run(vlab_module_t handle, void *user)
{
  struct mod_hist_info *info = (struct mod_hist_info *) user;
  vlab_buffer_t inbuffer, outbuffer;
  long *indata, *outdata;
```

```
  int i;
  for(;;) {
    *info->count = 0;
    for(i=0; i<info->range; i++)
      info->aggregates[i] = 0;
    outdata = vlab_prepare(info->out, &outbuffer, 0);
    vlab_updateParameter(info->thresholdref);
    while(*info->count <= *info->threshold) {
      indata = vlab_read(info->in, &inbuffer, 0);
      info->aggregates[*indata] += 1;
      vlab_release(inbuffer);
      ++(*info->count);
      vlab_updateState(info->countref);
    }
    for(i=0; i<info->range; i++)
      outdata[i] = info->aggregates[i];
    vlab_write(outbuffer);
  }
}

struct vlab_specification mod_hist = { 1, "hist", &initialize, &run };
```