1999

MEADE: A Modular, Extensible, Adaptable Design Environment for ASIC and FPGA Development

Gary Spivey
George Fox University, gspivey@georgefox.edu

MEADE: A Modular, Extensible, Adaptable Design Environment for ASIC and FPGA Development

Gary Spivey

National Security Agency
Ft. Meade, MD
spivey@nsa.gov

ABSTRACT

We present MEADE, a Modular, Extensible, Adaptable Design Environment. MEADE has been developed to answer the need for an adaptive design framework for encapsulation of Computer Aided Design (CAD) tools and management of the massive amounts of data associated with the design process. Other frameworks have existed but lacked the critical open source requirement that enables rapid adaptation to a rapidly advancing design methodology. While the initial application and development of MEADE is targeted toward ASIC and FPGA design, the MEADE engine can be easily adapted to abstract any procedural application.

MEADE allows the definition of procedures, which are defined as some sequence or flow of actions, which can be performed by potentially multiple different agents. With this system, design methodology management is specified in the procedures. Tool interoperability is handled by the action definitions. The unique agents perform tool interchangeability (the use of "best-in-class" tools). All details of procedure implementation are extended outside of the MEADE microkernel to the individual agent modules (Source code control, code builds, multi-user simulations, etc.). With an open, extensible system, the design community will be able to integrate specific design flows and account for site-specific variances. Additionally, new CAD tools can be rapidly integrated into a design flow for effective evaluation. It is believed that the simple modular interface and open-source philosophy will enable MEADE to succeed where other CAD frameworks have failed.

1 Introduction

As the semiconductor industry enters the "System on a Chip" age the management of Computer Aided Design (CAD) tools and design methodology is far from trivial. With the advent of multi-million gate ASIC designs we have seen an explosion in the number of distinct CAD stages, the amount of processing that the CAD tools perform, and the amount of data required to represent the design. Operating these CAD tools can be extremely difficult as there are many details to manage; tool invocation, run-time environment, error conditions, data translations, resource allocation, and data management are just some of the obvious issues.

In order to control the design process and manage the large amounts of data generated by it, CAD frameworks have been developed. A CAD framework can be defined as "a software environment that enables the coexistence and cooperation's of a variety of design tools. It provides a base on which tools can be developed and integration functions that shield end users from systems complexities … the primary function of a CAD framework is to facilitate tool interoperability and interchangeability" [1].

CAD frameworks have been developed in the past, and are being developed for the future. "There have not been many success stories to date in the design of and implementation of CAD databases, let alone CAD frameworks, although hundreds of millions of dollars have been spent trying to achieve this goal. We believe that the important reasons for this have very little to do with what designers are trying to build and are almost entirely concerned with the approach they take to the design and implementation of the system" [2].

It is our opinion that the chief feature missing from commercial and academic CAD frameworks is the general lack of an "Open Source" design philosophy. In a market as fast moving and technical as CAD, rigid, proprietary frameworks have been tossed aside with

reckless abandon. Some have been constructed with expectations of standardized data formats, others with expectations of proprietary tool-sets, and others looking toward the future with little present applications for design shops with a vested interest in software. As a result, many designers have hacked together some form of homegrown and commercial tools to describe a design flow. This is typically done in some amalgam of shell/perl/Tcl and other scripting languages in a less than object-oriented manner. This makes additions and modifications to an existing environment difficult (especially for those who did not develop the environment). With the rapid increase in the amount and type of CAD tools on the market, keeping up with the latest tools and integrating them into a local design environment proves to be an enormous challenge.

We are in the process developing and making public MEADE: A Modular, Extensible, Adaptable Design Environment. MEADE recognizes that in the process of designing hardware, many unique 'procedures' are performed many times. These procedures can be described as a sequence (or flow) of independent 'actions'. Furthermore, there mat be one or more 'agents' available to perform these actions. MEADE provides the mechanism to describe these procedures, actions, and agents in an object-oriented manner that is both easy to understand and easy to adapt. The agent performs most of the work done in MEADE and is, generally speaking, an encapsulation of some commercial tool.

MEADE can be viewed as a small operating system. MEADE is essentially a 'microkernel' that processes and controls the execution of these procedure/action/agent specifications. MEADE also provides additional 'kernel modules' to perform the basic tasks of the design environment. On top of this kernel are application modules that perform any variety of tasks within the MEADE environment. As previously stated, we are developing MEADE for the purpose of ASIC and FPGA development, specifically, development using the Verilog Hardware Description Language (hereafter referred to as simply Verilog). In order to facilitate this effort, we have provided kernel modules to control design management, source code control, Verilog module dependency checking and preprocessing, etc. On top of this subsystem, we have developed application modules for a number of actions (including certain simulation, synthesis, and analysis tools). Additional modules will be provided as new tools are integrated into the system. Much work has been done to ensure that the microkernel and kernel modules will not be overly intrusive on the architecture of these application agent modules. This is supported by extensive use of object-oriented design wherever possible.

With MEADE, design methodology management is specified in the procedures. Tool interoperability is handled by the action definitions. The unique agents perform tool interchangeability (the use of "best-in-class" tools). All details are extended to the application modules (Source code control, code builds, multi-user simulations, etc.). With an open, extensible system, the design community will be able to integrate specific design flows and account for site-specific variances. Additionally, new CAD tools can be rapidly integrated into a design flow for effective evaluation. It is believed that the simple modular interface and open-source philosophy will enable MEADE to succeed where other CAD frameworks have failed.

This paper discussed the design and construction of MEADE along with some of the more interesting applications. Section 2 provides the scope of the problem for which MEADE was developed. Sections 3 and 4 discuss the MEADE microkernel and kernel modules. Section 5 presents an overview of some of the MEADE application modules and is followed by a conclusion in chapter 6. Wherever references to specific perl modules or application programs are made, references are given as to where to locate them on the web (with the exception of CPAN modules [3]).

2 Scope of the Problem

From the perspective of the design engineer, the first step in the design process using Verilog is design entry. In modern design centers, this can be performed by an increasing number of tools (including gate-level schematics, graphical HDL tools, text based HDL entry, and now C or JAVA code as a front end that can be compiled to Verilog). Text based entry of the Verilog code is still the standard, but allowances must be made for other methods. The design is typically broken down into a number of smaller blocks that can be parceled out to different engineers on a design team. Designers will often develop 'test benches' that will test the modules that they are designing. Additionally, teams of 'verification engineers' are used to do design testing at a higher level.

All of this testing is done using Verilog simulators. There are interpreted Verilog simulators, native-code compiled simulators, and cycle based simulators. When the testing is being done, dumps of different simulation states are done that can be analyzed by a number of different analysis tools. There are also code 'linters' available for debugging prior to simulation.

When the code is functionally correct, it is passed through a 'synthesis' tool that converts the behavioral description into a 'netlist' representing a connection of cells in the target technology. This synthesis step can be generalized or highly specific using (typically) TCL scripts. After all of this the netlists go on to placement, layout, timing checkers, and then the results of this get fed back in to do simulation and synthesis all over again (sort of like the bill in the old Schoolhouse Rock video).

In addition to interfacing with all of these tools, methods need to be provided to manage the multi-user design. Obviously, some form of version control system needs to be provided. However, it is also necessary to mediate the interaction of these various design 'nodes' (a node represents a level in a block diagram that is generally relegated to one design engineer). In MEADE, the design nodes are handled by the version control system as independent units. In the course of a design, a 'golden' model will be developed that emulates proper system execution (this may be a very high level description). As the design progresses, nodes of this golden model may be replaced with more complete (and complex) nodes. It is desirable to be able to simulate the system with a variety of nodes under development and golden nodes.

MEADE first began, like most other attempts, as a method to handle some of these basic internal design flow issues – mainly automating redundant tasks. The initial version (then known as VDE – a Verilog Design Environment) was composed of perl, C-shell, and several makefiles. It was observed that the environment was rigid and difficult to modify. A second version was developed using the procedure/action/agent model. This was written in perl, with makefiles handling dependencies and build parameters. Agents were stored in individual perl modules. While an improvement, the procedure structure was still quite rigid and the lack of object-oriented design principles limited the effectiveness of tool integration and agent modification.

With more tools constantly being required in our design flow (and a lack of personnel to maintain the flow), it was deemed necessary to develop a more robust environment that would be easier to maintain. An added concern is the 'debatable' shift from UNIX to WinTel based tools and/or front-end's for ASIC and FPGA development. Perl has a large advantage over other scripting languages in its support for both platforms.

Thus, the MEADE project is being developed completely in object-oriented perl5 (to the extent that pTk is being used for the GUI, and cons will likely be included as a default for dependency checking and software construction, rather than the traditional UNIX make program). Additionally, Perl was chosen because of its widespread use in the community and dominant success as a glue language and text parser, two obvious requirements for developing a framework.

3 MEADE Microkernel

We refer to MEADE as a small operating system (hopefully without offending any real operating systems). The microkernel approach is taken where only the essential functions are represented in the microkernel and everything else is relegated to other modules to enhance adaptability. The microkernel consists of the MEADE object as well as other procedure, action, and agent class definitions and any subclassed objects created from these definitions.

3.1 Creating the microkernel

The MEADE microkernel is encapsulated in a MEADE object. This object is created by an external script, its procedures are created, and then the object is activated. An example of a startup script is given below. This method allows different home directories to be specified via environment variables as well as libraries and procedure files. The procedure file is simply a perl script of use statements for different procedures. The procedure file variable can be overridden so that a personalized set of procedures can be used.

```
#!/usr/bin/env perl
require 5;
use strict;
use vars qw($MEADE_HOME $MEADE_LIB
    $MEADE_PROCEDURE_FILE $MEADE_TYPE);
use Env;
use Carp;
use File::Basename;
# Establish run-time variables
BEGIN {
    $MEADE_HOME ||= "/usr/local/meade";
    $MEADE_LIB ||= "$MEADE_HOME/pm";
    $MEADE_PROCEDURE_FILE ||=
    "procedures::Meade_Procedures";
}
use lib $MEADE_LIB;
use lib ("$MEADE_HOME/pm");
use Meade;

my ($name, $path, $suffix) = fileparse($0,
    '\..+');
if ($name =~ /^g/) {$MEADE_TYPE = 'Meade::gui'}
else {$MEADE_TYPE = 'Meade::text'}
eval "use $MEADE_PROCEDURE_FILE";
```

```
my $meade = new Meade ( home => $MEADE_HOME,
                        type => $MEADE_TYPE,
              );
$meade->create_procedures;
$meade->activate;
```

3.2 The MEADE object

The MEADE object is the central object in the MEADE environment. It provides a linkage between all objects in the system and is generally accessible by any object (if need be).

3.2.1 MEADE Methods

The MEADE object defines a few simple methods, the most obvious of these being the 'new' constructor. MEADE make extensive use of the wonderful Class::MethodMaker module so that any variable that needs to be referenced outside of the object has an accessor method (if you don't use MethodMaker, you should …). The constructor for the MEADE object is presented as it is nearly identical to every other constructor method used in MEADE.

MEADE uses the two-argument form of bless to allow for inheritance. Note also the foreach loop that accesses all accessor methods passed in as arguments. (Where this clever little trick came from has long since been lost to me and I can't find it in any books or other code, but I am interested in discovering the source).

```
sub new {
    my $proto = shift;
    my %args = @_;
    my $class = ref($proto) || $proto;
    # Create a new object hash
    my $meade = {};
    # And bless is into this class
    bless $meade, $class;
    # If this is an Instance method
    if (ref ($proto) ) {
        # Copy all of the keys from the object
        foreach my $key (keys (%$proto)) {
            $meade->{$key} = $proto->{$key};
        }
    }
    else {
        # Otherwise, it is a Class method
        # set up defaults,
        $meade->home ('/usr/local/meade');
        $meade->type ('text');
    }
    # Optional keys from the argument hash
    # Basically, every variable is accessed
    # through a get_set method of the
    # same name (see the MethodMaker).
    # So, if the object can perform the set,
    # then delete that option and
    # perform the set
    foreach my $key (keys %args) {
       $meade->$key(delete $args{$key}) if
           $meade->can($key);
    }

    my $MeadeType = $meade->type;
    unless (eval "use base qw($MeadeType)") {
    return $meade;
}
```

The MEADE object also provides a class attribute, @Procedures. This attribute can be added to using the add_procedure method. At the beginning of every procedure file should be the line:

```
$MEADE->add_procedure;
```

The add_procedure method strips off the package name of the calling procedure and adds it to the @Procedures attribute. When the create_procedures method is called on the MEADE object, MEADE will iterate through the @Procedures variable and create procedure objects for each procedure that registered itself with MEADE. The MEADE object also contains an active_procedure method (which, uh … accesses the active procedure) as well as a simple execute method. The execute method does slightly more than just call the execute method of the active procedure. It also creates a 'blackboard' object and passes is to the procedure for use in parameter passing among agents.

3.2.2 The Blackboard Object

The blackboard object is a small abstraction from simply passing a hash reference. The blackboard constructor is similar to the MEADE object constructor - it differs only in that there are no default variables. While the blackboard specifies no accessor methods (the blackboard is simply accessed by reading and writing to named keys), it does specify a read and a write method.

The write is a two-argument method (key and value). This method simply writes the value to the key in the hash. However the read has a subtle twist. The point of the blackboard is to have a dynamic parameter space to pass between agents as they execute. However, there are times when an agent will want to read a value from the

blackboard and then modify it for its own internal use. In many of these cases, it is desirable to pass on the original value rather than the modified value. For this reason, when an ARRAY or HASH reference is read from the blackboard, a copy of the data will be returned. If the agent makes any modifications to the elements referenced by the ARRAY or HASH, the original blackboard will not be affected. A write must explicitly be done to the blackboard in order to save any desired changes.

```
sub read {
   my $blackboard = shift;
   my $key = shift;

   my $value = $blackboard->{$key};
   return $value  if (not ref $value);
   return $$value if (ref ($value) eq 'SCALAR');
   return $value  if (ref ($value) eq 'CODE');
   return $value  if (ref ($value) eq 'GLOB');
   if (ref ($value) eq 'ARRAY') {
       my @array = @$value;
       return \@array;
   }
   if (ref ($value) eq 'HASH') {
       my %array = %$value;
       return \%array;
   }
   return $value;
}

sub write {
    my $blackboard = shift;
    while (@_) {
        my $key = shift;
        my $value = shift;
        $blackboard->{$key} = $value;
    }
    return (1);
}
```

The blackboard provides a powerful ability for MEADE – data abstraction. In the rapidly changing world of CAD tools, it has been extremely limiting to define a way to represent data. Using the blackboard, agents can pass data of any representation between themselves in a growing, adaptable manner

3.3 Microkernel Modules

Beyond the MEADE object, the microkernel consists of procedure class definitions, action class definitions, and agent class definitions as well methods to process and execute any defined procedures. The command line interface and a graphical user interface, as well as all other elements of MEADE, the design representations and tool encapsulations are considered 'outside' of the MEADE 'microkernel'. This allows flexibility and adaptability in the development of new tool and procedure integration.

The MEADE object contains a reference to all available procedures and may also contain references to any extended elements (e.g. graphical interface windows). A procedure object specifies a list of actions and orders the execution of those actions. The action objects specify agent objects that are capable of performing the action. The agent objects are responsible for performing the action and contain option objects that can be modified to influence the agent behavior.

The MEADE object contains methods to initialize all modules and initiate the active procedure's execution method. As procedures include actions, and actions include agents, we will discuss them in a bottom up fashion.

3.3.1 Agents

The agent is the heart of the MEADE system. Agent objects perform all MEADE tasks beyond initialization and flow control. MEADE determines very little about the agent's setup. Other than the constructor method, the only non accessor methods provided in the agent class are the option setup methods (add_option, init_options, parse_options).

3.3.1.1 Agent Options

MEADE uses an object for each option and the agent maintains a list of its option objects. Options objects contain accessor methods for variables such as the option's name, accessor method, type, default, and information statements to be used help messages, status bars, or balloons. The option name is used by command line parsing.

Rather than supplying a variable for the option, an accessor method is provided to extend capabilities to the option access mechanisms. This allows greater flexibility in the dynamic setting of options. For instance, the option default can be a value, but can also be a subroutine reference. This is necessary, as the default for an option is often some derivative of dynamic run-time information (e.g. the default name of a data file is likely dependent on the name of the design object being operated on). By using a method rather than a variable, the value to set the option to can be checked and evaluated before being set in the case that the value is in fact a subroutine reference.

The option type method provides a higher level of abstraction than the standard Getopt option types.

Currently Getopt::Long is used, but the potential of using the declarative Getopt::Declare module is being investigated. The option class provides an option check method insures that the option can be set to the specified value. An option_get_set method was created using Class::MethodMaker that embeds this check in the get_set method of each option. The resulting accessor method looks like the following (for an option named 'name').

```
sub name {
  my $agent = shift;
  if (@_) {
    my $value = shift;
    my $new;
    my $option = $agent->option_hash->{$name};
    my $type = $option->type;

    # Check the new value to see if it
    # is code
    # If so, evaluate it
    if (ref ($value) eq "CODE") {
      $new = &$value($agent);
    }
    # Otherwise, simply set it
    else {
      $new = $value;
    }
    # Now check to see if the option is valid
    if ($option->check_option($new)) {
      $agent->{$name} = $new;
    }
    else {
      my $agent_name = $agent->name;
      my $error = "MEADE: agent $agent_name,
    option $name is of type $type and cannot be
    set to $new";
      croak ($error);
    }
  }
  $agent->{$name};
};
```

Options are initialized at MEADE startup by the agent initialization method and can then be operated on by the interface (GUI or command line).

3.3.1.2 Agent Module

The agent specifies accessor methods for its name, parent action object, internal option hash and a few other details for the agent. A standard agent module looks similar to the following…

```
package agents::AGENTNAME;

require 5;
use strict;
use Carp;
use base qw(agents::agent);
use Class::MethodMaker
    get_set => [qw ( somevar )];
# This is a special subclass of MethodMaker that
# creates get_set methods that
# will also do option type checking
use agents::OptionMaker
    option_get_set => [qw (debug infile)];

sub init {
    my $agent = shift;
    my %args = @_;

    # Optional keys from the argument hash -
    foreach my $key (keys %args) {
       $agent->$key(delete $args{$key}) if
         $agent->can($key);
    }
    # Add the options
    $agent->add_options;
    return(1);
}

sub add_options {
    my $agent = shift;
    $agent->add_option (
      name => 'debug',
      type => 'Toggle',
      default => 1,
      method => 'debug',
      info => 'Turn debugging on',
      );
    $agent->add_option (
      name => 'infile',
      type => 'ReadFile',
      default => sub {$agent->somevar . 'in' },
      method => 'infile',
      info => 'The input file',
      );
}

sub execute {
    my $agent = shift;
    my %args = @_;

    $agent->init_options
      if $agent->can('init_options');
    # Optional keys from the argument hash
    foreach my $key (keys %args) {
       $agent->$key(delete $args{$key})
         if $agent->can($key);
```

```
    }

    # Get the MEADE blackboard so that we can
    # write things to it. Namely, the
    # return value
    my $blackboard = $agent->blackboard;

    … execution specifics and tool
    encapsulations …

    $blackboard->write(status => $status_val);
}
```

The agent class provides the basic accessor and option methods. What is specified in this agent is the initialization, add_options, and the execute methods. The initialization routine is called at startup which in turn calls the add_options. This completely defines the parameters of the agent module. When it is time for execution, the execution method is responsible for option initialization and processing of incoming parameters (this can be done in any order as it is implementation dependent). The agent is also responsible for writing a return value to the blackboard status space for use in the procedure execution engine (actually, if the status space is empty, the procedure method will politely use the return value of the agent). This allows the procedure to make decisions about which action to perform next.

3.3.2 Actions

The MEADE action object is a simple abstraction that allows multiple agents to be interchanged to perform the specified action. In many actions there may be but one agent available to perform the action. In this case, the action is a useless level of indirection. However, as performance in MEADE is a relative non-issue (MEADE execution times will be dwarfed by the run times of the tools being "glued" together), this level of indirection is worth the cost. The large win of the actions occurs when multiple agents exist and the design engineer can select between the agents to use "best-in-class" tools for differing functions. For example, when simulating a Verilog circuit, an interpretive simulator (Verilog-XL) may be much faster on smaller modules while a native-compiled simulator (Synopsys VCS or Cadence NCV) will outperform the interpreted version on large RTL blocks. This ability to select different tools to perform the same function, and still appear the same to the user, is one of the chief functions of MEADE.

3.3.2.1 Action Module

The action class specifies a method to construct a new action object and a method to add an agent to the object's agent list. The add_agent method pushes the agent identifier onto the action's agent list and also sets the agent's action variable to the parent action. For the following code, assume that we have a generic action called "HOP" and we want to tell MEADE that there are three different agents that can "HOP", namely "GRASSHOPPER", "BUNNY", and "SCOTCH". The following init method would be called my the action constructor method.

```
sub init {
  my $action = shift;
  my %args = @_;

  use agents::GRASSHOPPER;
  use agents::BUNNY;
  use agents::SCOTCH;
  $action->add_agent(new agents::GRASSHOPPER());
  $action->add_agent (new agents::BUNNY());
  $action->add_agent (new agents::SCOTCH());
}
```

That is basically all that there is to the action module, a specification of agents. If desired, parameters can be passed in to the agents on creation as well.

3.3.2.2 Action Vectors

Additionally, the action object contains a vector object that can is stored in the action->vector method. The vector is a list of value, action object pairs. The purpose of the vector object is to provide a linkage from the current action to other action objects for flow control. The method used to obtain the next action in a flow is the vector->next_action method.

```
sub next_action {
    my $vector = shift;
    my $key;
    my $action;

    # See if the key was set on the method call
    if ($#_ != 0) {
      $key = 'default';
    }
    else {
      $key = shift;
    }

    if ( defined
       ($action = $vector->query($key))) {
       return $action;
    }
```

```
    elsif ( defined
       ($action = $vector->query('default'))) {
       return $action;
    }
    elsif ( defined
       ($action = $vector->query('error'))) {
       return $action;
    }
    return (undef);
}
```

The next action is queried using the passed in key value. If that does not exist, a default key is searched, followed by an error key. If none of these exist, then the action is returned undefined.

The vector object can be modified with add/delete methods. The object can also be queried for its current list of pairings, or queried for an action object given a value. How the action vector is used will be discussed in the following section.

3.3.3 Procedures

MEADE procedures specify a list of actions that may be performed in the procedure. Typically, the actions are in a sequence, but mechanisms are provided to allow for flow control. Actions can be initialized as required, enabled, or disabled. Actions that are not required can be enabled/disabled by the user via the MEADE interface for particular procedure executions.

3.3.3.1 Procedure Flow Control

MEADE provides a procedure class and specific procedures are subclasses. The procedure class provides a standard constructor method for a new procedure object (using the typical perl hash reference as an object). Accessor methods are provided to specify both a start action and to hold the last action created by the procedure. An add_action method is supplied that will build an action list and set a sequential execution order.

```
sub add_action {
    my $procedure = shift;
    if (@_) {
        my $action = shift;
    $procedure->start_action($action)
      if (!defined $procedure->start_action);
    # Check and see if there was an action
    # assigned before this one
    if ($procedure->last_action) {
        # set up the vector to branch to this
        # action in the default case
            $procedure->last_action->
               vector->add(default => $action);
    }
    $procedure->last_action($action);
    # Add it to the action list
    $procedure->add_action_list($action);
    # Set the actions procedure
    $action->procedure($procedure);
    return $action;
    }
    else {
    croak ("MEADE: An action argument is
    required in the add_action method\n");
    }
}
```

The above method simply checks for the existence of a previous method (last_method) and then, if one exists, sets its vectors default next action to be the current action. A create_action method is also provided that clears the last_action before calling add_action. This allows the user the opportunity to break the standard sequential flow.

If non-sequential flow control is desired, this must be instrumented on a procedure by procedure basis. Flow control can be accomplished by setting the vector of the action object. The procedure's execute method will sequence through the actions based on the value of the action vector. The action vector is an object that maintains return value, next action pairs. In this way, the return value from the agent is passed back to the MEADE engine, which then obtains the next action to execute. The agent for this action is then executed. By setting vectors for multiple return values, branches and loops can be accomplished.

3.3.3.2 Procedure Execution

The standard procedure execution method first determines the start action. If that action is enabled, it will then determine the agent to perform the action and the method call to invoke the agent's execution. The blackboard status is cleared and the agent is then invoked. Upon return, the blackboard is checked for a status value and, if it is not set, the return value is checked. The resultant status value is then used to query the actions vector to obtain the next action to operate on. When this action is returned, the cycle begins again just as it did for the start action. This will continue until the next action is undefined.

```
sub execute {
  my $procedure = shift;
  my %args = @_;

  # Optional keys from the argument hash
  foreach my $key (keys %args) {
      $procedure->$key(delete $args{$key})
```

```
        if $procedure->can($key);
  }
  my $action = $procedure->start_action;
  while ($action) {
    my $status;
    if ($action->run ne 'off') {
      my $agent = $action->active_agent;
      my $method = $agent->method;
      my $blackboard = $procedure->blackboard;
      $blackboard->write(
          action => $action,
          agent => $agent,
      );
    $status = $agent->$method(
        blackboard => $blackboard
     );
    $status = $blackboard->read('status') ||
        $status;
    # Clear the status for the next run
    $blackboard->write( status=> undef);
    $action = $action->vector->
                next_action($status);
    }
    else {
      $action = $action->vector
              ->next_action($status);
    }
  }
}
```

3.3.3.3 Procedure Module

The following is an example of a normal sequential procedure. For example purposes, we will work with a procedure called 'Travel'. The procedure has three actions that are to be performed in sequence. These actions are 'HOP', 'SKIP', and 'JUMP'. In this example, the HOP action is required, with the SKIP action being normally enabled, and the JUMP action defaulting to the disable state.

```
sub init {
    my $procedure = shift;
    my %args = @_;

    # Set up procedure parameters
    $procedure->name('Travel');
    $procedure->info('This is the info
        for the Travel Procedure');
    # Optional keys from the argument
    foreach my $key (keys %args) {
       $procedure->$key(delete $args{$key})
           if $procedure->can($key);
    }
    # Setup the actions for the procedure
    use actions::HOP;
    use actions::SKIP;
    use actions::JUMP;
    $procedure->add_action(new actions::HOP (
        message => 'HOP Item',
        run => 'required',
        ));
    $procedure->add_action(new actions::SKIP (
         ));
    $procedure->add_action(new actions::JUMP (
        run => 'off',
         ));
}
```

This is the normal model for procedure creation with sequential actions. If a flow control is desired, the vector model can be used.

```
    use actions::HOP ;
    use actions::JUMP ;
    use actions::SKIP ;
    my $hop= $procedure->add_action (
        new actions::HOP ( ));
    my $skip = $procedure->add_action (
       new actions::SKIP ( ));
   $skip->vector->add('loop',$hop);
    my $jump = $procedure->add_action (
        new actions::JUMP ( ));
```

In addition to the normal sequential flow (automatically added to each the action vectors using the add_action method), we have added another vector to the SKIP action vector with the key 'loop'. In this case, the SKIP action is set to loop back to the HOP action whenever it returns the value 'loop'. The agent performing the SKIP action sets this return value. Using this method, any form of flow control can be modeled.

4 MEADE Kernel Modules

The MEADE kernel modules are modules that are considered as part of the specific design environment (or operating system for the analogy), but not necessarily essential to MEADE itself (e.g., the interface, design management portion of the GUI, node management modules, design creation modules, etc.). These definitions are not considered as rigid, but allow a method to view the importance of the given modules. For instance, the design creation module scheme must be more foundational then the modules that operate on the design itself. In Verilog coding, aspects controlling the tools must often be embedded in the code itself (e.g.

frequency of the simulation, file I/O parameters). It is necessary to define some standards that will be supported by the rest of the system. As these standards are more germane to Verilog than Perl, they are not discussed in this paper. But suffice it to say that beyond the initial procedure/action/agent model, a few more guidelines must still be enforced for effective framework operation.

MEADE provides these support modules as either procedures or actions themselves. In this way, MEADE does not compromise its basic operation to account for specific design environment implementations (again, recognizing the need for future adaptability). So, all design object creation and manipulation are handled by external 'kernel' modules and are not defined by the MEADE object. This should allow for a more open exchange of modules between systems for different purposes (e.g. it should be quite simple to implement a graph drawing algorithm from a MEADE implementation focused on software development into the MEADE implementation for ASIC/FPGA development).

4.1.1 MEADE Interfaces

MEADE is currently invoked using either a command line or graphical interface (although there are plans to extend it to a browser interface as well). In any interface, a procedure is specified, actions are enabled/disabled, agents are selected, and options are set. Then the MEADE object is asked to execute the procedure as previously defined.

4.1.1.1 Text (command line) interface

The MEADE command line is first parsed by a special MEADE interpreter and then packaged up for each agent to parse individually (the default class method uses Getopt::Long). The first argument on the command line is the name of the class of the procedure to run (if that does not match a procedure, the name is formed into a class with the assumption that the procedure class name is of the type 'procedures::name'). The argument list is then parsed looking for options that either enable (-action_name) or disable (-noaction_name) the actions that make up the procedure. Additionally, agents can be specified by equating them with the action using the '=' (-action=agent). Any other arguments are packaged up and sent to each individual agent when it is time for the agent to execute. If options are to be designated for a particular agent, they can be included with the agent specification using a quoted field for the agent (-action="agent –opt1 –opt2…"). Any options specified in this manner will be delivered to that agent and that agent alone.

When an agent parses the options, it is up to the agent to handle any unknown options and potentially pass them on to the tool that the agent may be invoking. The agent class method for parsing saves all of the unused arguments in 4 unique arrays (global and unknown options, specific and unknown options, global an non-option arguments, specific and non-option arguments). Error checking is performed on agent names as well as option types.

4.1.1.2 GUI interface

The graphical user interface (GUI) for MEADE is described wholly using pTk. The essential portion of the GUI contains a main window that houses a procedure window. The procedure window contains a menu bar with a procedure menu button, help, quit, and execute buttons.

MEADE has an 'isgui' method that is accessed by each procedure as a beginning compile step. If MEADE is a GUI, then the procedures load any pertinent GUI information, specifically a procedure entry into the menu button. The entry takes the form of a label and a path (with other arguments indicating variant options, such as a unique method to redraw the procedure window). Rather than simply providing a label, the path specifies submenus of the procedure menubutton in which to place the procedure label. In this way, procedures can be grouped together in submenus of like procedures (e.g. Different version control procedures may have an identical path name (e.g. 'VCS') with differing label names (e.g. 'update', 'check out'). Procedures may have multiple menu-item entries.

The procedure class provides a method for drawing the procedure window. Currently it is extremely limited in that it only draws a sequential path of default action items, but provision is being made to handle cyclic directed graphs so that both branching and flow control can be displayed. The action items are constructed by each action using an action method. The action items contain a selection menu of available agents, an enable/disable switch (if the action is not required), and an option button. An option window (created from Tk::Getopt) is attached to the option button. The method for creating the option window is provided by the agent class and each agent has its own option window. When a new agent is selected, that agent's option window appears in the place of the former agent's window.

Using the above elements, the MEADE window is fully defined and allows for the basic operation of MEADE. In the specific instance of our Verilog design environment, another subwindow of the main window is created to handle design information input. This window

is still under construction and will enable the user to select design nodes to operate on as well as objects within those nodes. This technique demonstrates the modularity of MEADE and its independence of various formats as whole interfaces can be easily added into the environment by external modules. Again, the need for future adaptability is viewed as being of paramount importance.

4.2 Design Objects

In our prior Verilog Design Environment, data objects were referenced off of a module name. The name was passed into the environment and then all necessary references were built using that name (e.g. if the passed in name was adder, then the Verilog file would be adder.v, the constraint file would be adder.con, etc.). This model proved to be very limiting in that not only was the design limited in its naming scheme, but there was no notion of a design object other than the name itself. Nothing could be tagged to stay with the design object. For instance, if a particular module required a different simulation filename, there was no way to save that information with the object.

MEADE uses another blackboard object to define each data object. This object can be saved and read using the Perl Data::Dumper module. When a MEADE procedure exercises an agent to obtain a design object, the blackboard can be read in from a file. Further agents can use this information to set options and/or determine courses of action.

4.3 Help System

In any system that attempts to do actions for users, documentation is important in both enabling the user to use the tool as well as informing the user of what has been done. MEADE uses existing browser technology to handle large documentation needs and also has the capacity to generate html files from Perl scripts using the pod2html utility. Additionally, a publicly available Verilog2html utility will be incorporated for system level documentation of the underlying Verilog code [4]. This utility is also written completely in Perl.

5 MEADE Application Modules

In the case of the Verilog Design Environment that MEADE was constructed to facilitate, we have developed several application level “programs” and “wrappers” to effectively manage the design process and perform tool encapsulation. Details of some of the more interesting are given here.

5.1 Data Management

When designing with an HDL, the core of the design process is the HDL description, in our case, the Verilog file. MEADE provides methods to create new modules and test benches from existing templates, with the appropriate substitutions also being performed by Perl. Data is kept in a source directory or group of directories (as defined by the user). In order to make MEADE an effective tool for encapsulation of many of the Verilog tools, some elements of the Verilog code and test bench must expect to receive configurable information from command line invocation so that the more mundane details can be automated.

Therefore, we have developed some default templates as well as some Verilog interface code (via the Verilog Programming Language Interface , PLI) to be able to pass parameters to into the Verilog simulation or synthesis tools (these parameters include items like frequency, test data offsets, output file names, constraint file names, etc.) We have also provided the ability to copy, and delete modules in a way that maintains data stability in MEADE.

In addition to the Verilog templates, modules are being developed to support other forms of design specification (including graphical HDL’s and Java and C source code descriptions). All of these input forms get mapped to a Verilog file and MEADE must also provide a way to get data into the Verilog files generated by these methods. Again, the choice of Perl as a programming language enables us to perform this requirement with much greater ease than any other programming language.

5.2 Source Code Control

Source Code Control may be performed by a plethora of tools, some of them good, some of them bad, one of them free. For this reason (being free), MEADE is initially implementing source code control using CVS [5]. There is at least one source code control designed specifically for Verilog (SOS [6]). In the future MEADE will look to provide additional agents/actions/procedures for performing many of the same functions as CVS using this proprietary tool.

The golden directories will be easy to maintain using CVS updates of specified design nodes. Multi-user design teams will be protected at the node level, and most importantly, it will be easy for engineers to actually

use source code control. This reason alone may be enough to justify the entire environment.

5.3 Software Builds

In earlier versions, the UNIX make utility was used for maintaining dependencies and performing software builds (simulations and synthesis in the case of our environment). MEADE has decided to stay in a perl environment and use cons as the default software construction utility for the Verilog design environment. With cons, build files can be easily configured from Perl and the hierarchical abstractions of the design are easily supported. Additionally, a cons 'agent' can dynamically build inside the MEADE environment much better than an external makefile.

MEADE provides two different levels of building. The first level is the preprocessing step that is almost required to make up for deficiencies in the Verilog language. MEADE uses EP3 [7] (an extensible Perl preprocessor) to perform Verilog preprocessing. EP3 allows users to define their own directives for the preprocessor and this has enabled many powerful abstractions for the Verilog language. One example is the construction of a signal directive that allows signals to be inserted once into a Verilog file and test bench and have the appropriate (differing) substitutions made in about twelve different locations in the file and its test bench. This has provided a large productivity gain for our engineers in the early stages of Verilog code development.

The next level of building is the dependency checking on included modules prior to simulations. In order to accomplish this task, the Verilog code must be parsed to determine included modules. We will be employing the Rough Verilog Parser (RVP) [8] to accomplish this task. RVP is another utility written completely in Perl (do you see a pattern developing here?). Using RVP in conjunction with cons, MEADE can assure that the entire design path of a Verilog module is up to date.

5.4 Tool Encapsulation

There is a large assortment of commercial CAD tools used in the design process. We will discuss elements of the major ones here.

5.4.1 Simulation

As discussed earlier, there are different types of Verilog simulators. One function of the design environment is to enable the user to effortlessly switch between the different types of tools to take advantage of a given tool's individual strength. Verilog-XL from Cadence is an interpretive simulator. Interpretive simulators can be extremely fast for small modules as there is no code compilation step. However, as the modules grow in size, a native-code compiled simulator begins (such as VCS from Synopsys) begins to heavily outperform the interpretive simulator for behavioral HDL simulations. There are also issues of functionality with other tools that require certain functions to be done in one or the other simulators.

MEADE allows the designer to pick which agent (and thus simulator) to use for a Verilog simulation. New agent modules can be easily added to support different simulators as a design-shop acquires them. As the community development of MEADE accelerates, the ability to download MEADE agent modules for different simulators will allow designers the ability to compare newer simulators with great ease, and thus effectively evaluate tools in a timely manner with minimal effort.

5.4.2 Analysis

Analysis tools are used to debug Verilog simulations by providing waveform viewers, source code steppers, and other such tools. Analysis tools present an interesting challenge to MEADE in that the tool may be required for multiple steps in a simulation procedure. For instance, the tool must set up various parameters to be used in the Verilog simulation in order for its dumpfile format to be used. If the simulation procedure enables a post-simulation analysis action, another call to the tool must be made to activate the analysis environment. MEADE meets this challenge by allowing agent modules to have different execution methods, and providing the ability to tie actions agent choices together. When a user views the simulation procedure, they might see two distinct actions, analysis setup, and analysis, which are being performed by the same agent, but different methods of that agent. When selecting a different agent for one of them, the other will automatically change as well. In this way, a single agent change can be tied to effectuate other agent changes.

Another challenge presented by the analysis environment is the difference between interactive and post simulation analysis. While a subtle difference to the user, the method of invocation is vastly different by the environment. In the case of the interactive simulation, the analysis tool must be started first and the simulation initiated from within the tool. The MEADE environment is capable of discerning these differences and making the difference appear as subtle as the user envisions.

5.4.3 Synthesis

Synthesis poses a more difficult challenge to a framework because of the sheer complexity of the dominant ASIC synthesis tool, Synopsys Design Compiler. One of the main design goals for MEADE was the desire to lend and not force structure. This is nowhere more evident than when performing synthesis. Novice synthesis users require robust defaults to handle most of the synthesis process, enabling designers to perform useful design steps while also familiarizing themselves with the tool. More experienced designers want the ability to freely modify scripts to control the synthesis flow. MEADE agents account for this and with the data object model can allow a wide variety of user specifications for a given synthesis run. Things such as where to place and when to load constraint files become highly adaptable within the context of the Perl agent. Ultimately, we anticipate that the synthesis modules will be the most 'adapted' portion of the MEADE system as many design shops have there own methods for performing synthesis. Because of MEADE's inherent adaptability, radically altering the format of the synthesis procedures should have little no effect on the usability of other MEADE modules.

5.4.4 Design Management Methods

Design management methods single module simulation and synthesis are required in an advanced design flow. As the design grows and pieces are being put together, a mechanism is required to facilitate this design integration. MEADE provides a framework for integrating design pieces together in a seamless, interchangeable fashion. By breaking the design down into its hierarchical components (or nodes), MEADE defines procedures to dynamically define design node connectivity and allow nodes to be interchanged. This is extremely useful when simulating a developing node in the context of the golden models. When a design node is verified as working with the golden models, it can then be considered golden as well. Multiple users can verify different nodes independently using MEADE's interchangeable design node structure.

6 Conclusions

While necessary to develop a design framework to better serve our local design center, we believe that and Open-Source framework serves in the better interests of our design center as it will enable other designer to contribute to the development and extension of MEADE. Additionally, vendors and/or users will be able to provide methodologies for running tools that will allow rapid integration and evaluation of tools into an existing design environment, without the need for reworking the entire design environment to accommodate the new tool.

MEADE's use of Perl as a development language provides portability to a variety of platforms. Perl is also widely understood by the design community and allows design flow engineers a familiar environment in which to adapt MEADE to their specific users. MEADE's use of object-oriented principles facilitates this adaptation by locating areas of change within objects rather than across procedural subroutines.

We believe that the MEADE design will be very advantageous in the development of an Verilog framework, and is readily extensible to support other HDL languages and eventually other design flow specifications in the design and development of ASIC's and FPGA's. Furthermore, the MEADE microkernel/kernel has been formed independent of any reliance on the purpose of creating microchips and can be easily built upon to support any procedural environment, including software development.

7 References

[1] D. Harrison, A. Newton, R. Spickelmier and T. Barnes, "Electronic CAD Frameworks", *Proceedings of the IEEE Vol. 78, Issue 2*, pp. 393-417 (1990).

[2] T. Scallan, "CAD Framework Initiative – a User Perspective", *Proc. 29th DAC*, pp. 672-675 (1992).

[3] http://www.cpan.org

[4] http://abrizio.com/v2html

[5] http://www.cyclic.com

[6] http://www.cliosoft.com

[7] G. Spivey, "EP3: An Extensible Perl PreProcessor", *Proc. International Verilog HDL Conference and VHDL Users Forum 1998*, pp. 106-113 (1998).

[8] http://abrizio.com/v2html/rvp.html