virtutech

DML Tutorial

Simics Version	3.0
Revision	1406
Date	2008-02-19

Contents

Chapter 1

Introduction

DML (short for *Device Modeling Language*) is a C-like programming language for writing device models for Simics using Transaction Level Modeling (TLM). Although device models can be written directly in C, using DML simplifies the structuring of the code, which makes it much more readable and mainatainable, and reduces the risk of making errors.

The DML compiler is called `dmlc`. It translates a device model description written in DML into C source code that can be compiled and loaded as a Simics module. The output of `dmlc` is a set of C source and header files that can be compiled in the same way as a hand-written module would be. See the *Simics Programming Guide* for details on writing Simics modules in C.

The DML language and compiler is described in detail in the *DML Reference Manual*. See also the *Device Modeling Guidelines* document for a more high-level perspective on how to write device models.

1.1 Requirements

The DML compiler and standard libraries, the documentation, and the example devices, are provided by the *DML Toolkit* package, which is distributed separately from the main Simics package.

To run `dmlc`, and to build a device model generated from DML, it is also necessary to have a Simics Model Builder license and a working Simics build environment; see the *Simics Programming Guide* for details.

1.2 Installation

The DML Toolkit requires that you already have a Simics installation; cf. Section 1.1. In the rest of this document, we will refer to the file system path of the root directory of your Simics installation as `[simics]`, and to your "home" directory as `[home]`.

How to set up a working Simics build environment is described detail in the *Simics Programming Guide*.

Note that to develop Simics modules on the Windows platform, you need to work in a Cygwin environment (`http://www.cygwin.com/`), with the default packages of the

Cygwin "devel" category installed, and the separately installed MinGW compiler suite (`http://www.mingw.org/`); see the *Simics Installation Guide for Windows* for more information. You also need to ensure that Simics isn't installed in a directory whose name contains any spaces, since this will confuse the build scripts.

The DML Toolkit package can be installed at the same time as the Simics base installation, using the normal install script. See the *Simics Installation Guide* for details.

1.3 Emacs DML Editing Mode

The Emacs (`http://www.gnu.org/software/emacs/`) extensible editor is the first choice for many programmers, and the DML Toolkit includes a customized mode for editing DML source files. The DML mode is an extension of the standard Emacs editing mode for the C programming language.

The source file `dml-mode.el` for the DML mode can be found in the `[simics]/scripts` directory. To use it, add the following lines to your emacs configuration file (usually `[home]/.emacs` or `[home]/.xemacs/init.el`):

```
(setq load-path (cons "[simics]/scripts" load-path))
(autoload 'dml-mode "dml-mode" "DML mode" t)
(add-to-list 'auto-mode-alist '("\\.dml\\'" . dml-mode))
```

(you may need to replace the text `[simics]` in the above with the full path to your Simics installation). After restarting Emacs, the DML mode should be automatically enabled when you open a file whose name ends in "`.dml`".

For more information, run the command `M-x describe-mode` in an Emacs buffer using the DML mode, or read the "Editing Programs" section of the Emacs documentation

1.4 Example Device Models

Two example device models are included in the DML Toolkit package: an AM79C960 Ethernet card and a DS12887 real-time clock. The source code for the device models can be found in the directories `[simics]/src/devices/AM79C960-dml` and `[simics]/src/devices/DS12887-dml`, respectively.

There are also two Simics start scripts that can be used to run the example devices in a simulated x86 machine. The start scripts are named `enterprise-AM79C960-dml.simics` and `enterprise-DS12887-dml.simics`, and can be found in the `[simics]/targets/x86-440bx` directory.

Note that to be able to test run the example devices using the supplied scripts, your Simics installation must include the simulated x86 machine `enterprise` (the directory `[simics]/targets/x86-440bx` must contain other files than the two mentioned above).

For further details about the example models, see the *Simics Programming Guide*.

Chapter 2

Getting Started

2.1 A Simple Device

The following DML program is an example of a "smallest device that does anything interesting":

```
// A very simple device
dml 1.0;
device simple_device;
import "io-memory.dml";

bank b {
    parameter function = 0;
    register r0 size 4 @0x0000 {
        method read() -> (result) {
            log "info": "Hello, bus!";
            result = 42;
        }
    }
}
```

This models a memory-mapped device with a single 32-bit (4-byte) register at offset 0, which upon a read access will return the value 42 as the result of the operation, and as a side effect print a log message with the text "Hello, bus!" to the Simics console.

To compile this program, you first need to set up a Simics workspace, using the `workspace-setup` script (see the *Simics Programming Guide* for details). In order to create DML skeleton files for your new device, you need to pass the flag `--device=`*modulename* to the script, as follows:

```
$ [simics]/bin/workspace-setup --dml=simple_device [home]/workspace
```

(If your workspace directory already exists and you just want to add a new module, you also need to specify the `--force` flag.) On Microsoft Windows, using Cygwin, you should use

the script called `[simics]/bin/workspace-setup.bat` instead; also, make sure you have the default packages of the Cygwin "devel" category installed, as well as a working MinGW installation, *before* you try to set up the workspace; see Section 1.2 for details.

You will now have a directory `[home]/workspace` containing (among other things) a GNU Makefile and a subdirectory named `modules`, which is where your modules are located. Open the generated skeleton file `[home]/workspace/modules/simple_device/simple_device.dml` in your favorite text editor, and modify its contents to look like the above program. (The main DML source file name should be the module name plus the extension `.dml`. Although they are usually the same, the module name does not have to be the same as the device name; a module could potentially contain more than one device.)

Now, go to the `[home]/workspace` directory and run GNU `make` (possibly named `gmake`, depending on your system). By default, this builds all your modules.

```
$ cd [home]/workspace
$ ./simics targets/vacuum/vacuum.simics
```

When Simics has started, you should be prompted with "`simics>`". Congratulations, you now have a loadable Simics module! You can exit Simics by typing the **quit** command, or **q** for short.

```
simics> q
```

2.2 Loading And Testing

To test your new module, start Simics and load the example machine configuration `vacuum`, as follows (from the `[home]/workspace` directory):

```
$ ./simics targets/vacuum/vacuum.simics
```

The `vacuum` machine is a minimal configuarion that contains only a clock and a memory space, called **phys_mem**, with some RAM mapped at addresses `0x10000000` to `0x1fffffff`.

At the Simics prompt, type the **list-modules** command to see if your module appears on the list:

```
simics> list-modules
```

Som of the modules in the list will have `Loaded` printed in the second columnt, but not your new module. You can load it using the **load-module** command:

```
simics> load-module simple_device
```

If the module was successfully loaded, it should now be marked as loaded in the output from by the **list-modules** command.

You can also try the **list-classes** command to see if your device apears in the list of available models. It should appear in this list even if you have not loaded the module yet. Normally, modules are loaded automatically when you try to use them, so the **load-module** command is mostly used for testing.

To test the new model, we first of all need to create an instance of the device class. This can be done with the following command:

```
simics> @SIM_create_object("simple_device", "dev1", [])
```

(the @ operator on the Simics command line executes the rest of the line as a Python expression), which uses the Simics API directly to create a new object named **dev1**.

To be able to communicate with the device object, it must be mapped into the physical memory space of the machine. This can be done using the ⟨**memory-space**⟩**.add-map** command.

```
simics> phys_mem.add-map dev1 0x1000 0x100 0
```

The first argument is the device to map, in this case our newly created object **dev1**. The following arguments are the base address within the memory space and the size of the mapped range. The fourth argument is the *function number* which is used by the device model to choose which register bank will handle the accesses to this mapping. In this example, we have a mapping from the range `0x1000-0x10ff` (0x100 bytes) to the range `0-0xff` of function 0 (bank `b`) of the device object **dev1**.

We can now try to read from register `r0` of our device. (Since it has internal offset 0, it will correspond to address 0x1000 of the physical memory.) Use the ***get*** method of the memory space to simulate a read operation:

```
simics> phys_mem.get 0x1000
```

(do not use the @ operator here). You should see the following output:

```
[dev1 info] Hello, bus!
42
```

The first line is the info message printed by the `log` statement in the ***read*** method of `r0`, and the second is the value that was returned by ***get***. The device is working!

2.3 Scripting

The file `targets/vacuum/vacuum.simics` (the one you specified as an argument when starting Simics in the previous section) is a Simics script; it contains the same sort of commands that you enter on the Simics command line.

To avoid having to enter the same commands again and again when you restart Simics, open the file in your editor and add the necessary commands from the previous section (i.e., creating the device object, and mapping it into the physical memory) to the end of

the script. Now, each time you start from the `vacuum.simics` script, your device will be automatically loaded and ready to test.

2.4 Logging

DML has direct support for writing log messages to the Simics logging facility, through the `log` statement. The most important logging concepts are the *type* and the *verbosity level* of the message, where the most common message types are `"info"` and `"error"`. The verbosity level is a number between 1 and 4, where 1 is used for important messages that should always be displayed, and 4 is used for messages that should only be printed when verbose logging has been requested. For error messages, the level is always 1. By default, Simics only displays messages of level 1 on the console.

In the previous example, no level was provided, which will make it default to 1. To set the level of a message, add it after the type string, but before the colon, as in:

```
log "info", 2: "This is a level 2 message.";
```

To change what messages are displayed, use the **log-level** command.

```
simics> log-level 4
```

Now make a memory access to the device, as before:

```
simics> phys_mem.get 0x1000
```

This time (apart from the "Hello bus!" message), you should see an info message saying something like `"Inquiry read from register b.r0"`. This is logged by the built-in code that handles register read accesses, and messages like this can be very useful when debugging a device model.

2.5 Configuration Attributes

A Simics configuration consists of a machine description and a few other parts, and it is divided into a number of *configuration objects*. Each device instance in the configuration is represented by such an object. Any Simics configuration object has a number of *attributes*. An attribute is a named property that can be read or written using the Simics API. The value of an attribute can be an integer, a floating-point number, a string, an object reference, a boolean value, a list of values, or a mapping from values to other values.

Attributes are used for several related purposes, but the most important uses are for *configuration* and *checkpointing*. All the internal state of a device object must be available through the attributes, so that a checkpoint of the current state can be saved by reading all the attributes and storing the values to disk. By reloading a configuration and setting all attributes from the previously saved checkpoint, the states of all devices can be restored to the checkpointed state and simulation can continue as if it had never been interrupted. When

creating a new configuration, some of the state must be given an explicit initial assignment, which makes those attributes also part of the configuration. There may also be attributes that are not part of the state, in the sense that they don't change during simulation. Instead, they control the behavior of the model, such as buffer sizes, timing parameters etc. Those configuration attributes can generally not be modified once the object has been created.

Attributes can also be used as a simple interface to an object, e.g., for inspecting or manipulating the state for debugging purposes.

A DML device model usually defines a number of attributes. By default, each `register` defines a corresponding attribute that can be used to get or set the register value, but more attributes can be defined by explicitly declaring them in the DML source. In our example above, for the register `r0` of bank `b` there will be a device attribute named *b_r0* (note the underscore).

2.5.1 Using Register Attributes

The ***get*** and ***set*** methods of a register are different from its ***read*** and ***write*** methods in that they do not simulate a hardware access (which often triggers side-effects in devices). Instead, they offer a direct way of reading or modifying the value of the register.

We can access the attribute from the Simics command line. Continuing our example from the previous sections, enter:

```
simics> dev1->b_r0 = 17
```

and then enter

```
simics> dev1->b_r0
```

which should print the value 17.

However, if we make a new memory access, as before:

```
simics> phys_mem.get 0x1000
```

we still get the message "Hello, bus!" and the value 42. But entering **dev1->b_r0** once again still returns 17. What is going on?

The answer is of course that we had hard-coded the ***read*** method to always return 42, no matter what. But this does not affect the behavior of ***get*** and ***set***, or the ***write*** method. Let's try to make a write access:

```
simics> phys_mem.set 0x1000 0xff
```

Entering **conf->dev1.b_r0** now prints the value 255 as expected. You can change the line

```
result = 42;
```

in the program to:

```
result = $this;
```

(note the `$` character), recompile, and try the same accesses again to check how a normal register would behave. Then change the code back to return 42 for the rest of this section.

It is in fact often useful to create registers which either return a constant (usually zero), or return a value that is computed on the fly. For such registers, it is unnecessary to allocate memory for storing the value inbetween calls. The allocation can be turned off by adding the following line to the body of the register:

```
parameter allocate = false;
```

note, however, that doing so makes it necessary to implement your own versions of the ***get***, ***set***, ***read*** and ***write*** methods. (Try adding the above line to your code, recompile, and run the example again. Note the messages you get when you try to access the register in the various ways we have shown.)

A full implementation of such a "synthetic" constant register could contain method definitions like the following:

```
method write(value) {
    /* do nothing */
}
method get -> (value) {
    value = 42;
}
method set(value) {
    if (value != 42)
        throw;
}
```

Try adding them (to the body of the register), recompile, and run the example again.

The standard library file `utility.dml` contains several pre-defined templates for common implementations such as this. To use it, add the declaration "`import "utility.dml";`" to your source file. The constant register can now simply be implemented as follows:

```
register r0 size 4 @0x0000 is (constant) {
    parameter value = 42;
}
```

or, if you still want to get a log message for each read access:

```
register r0 size 4 @0x0000 is (constant) {
    parameter value = 42;
    method read() -> (result) {
```

```
            log "info", 1, 0: "Hello, bus!";
            result = $value;
        }
    }
```

2.5.2 User-defined Attributes

It is sometimes useful to have device attributes that are not associated with any register. If we just want the attribute to behave as a data field, which stores a value of a simple data type such as `int32` or `bool`, we only have to specify the parameter `allocate_type`, as follows:

```
attribute foo "an integer attribute" {
    parameter allocate_type = "int32";
}
```

(note that the data type must be specified as a string). Try adding this code to your device (either before or after the `bank`), recompile and rerun the example. Enter the following command:

```
simics> help attribute:dev1.foo
```

This prints some information about the attribute. Note that the descriptive string you specified in the program is included in the online documentation.

You can now experiment with setting and getting the value of the attribute; e.g., entering

```
simics> dev1->foo = 4711
simics> dev1->foo
```

should print 4711.

If it is important that other parts of the device are updated whenever the value of the attribute is modified, the method ***after_set*** can be overridden to perform such updates. (By default, it does nothing.) For example:

```
method after_set {
    log "info": "Someone updated the attribute!";
}
```

Add this method to the body of the attribute, recompile and restart Simics, then try setting and getting the value of the attribute.

If you want the attribute to do things differently, such as not store the value between calls, or use a more complex data type, you need to do more work on your own, instead of using the `allocate_type` parameter; see the *DML Reference Manual* for details.

Chapter 3

Using C code and the Simics API

Since data types and method bodies in DML are written in an extended subset of C, you can easily call C functions from DML by using normal function call syntax, as in `f(x)` (rather than using the `call` or `inline` keywords to call DML methods). For example, the following is a legal DML program, based on the example in Section 2.1:

```
dml 1.0;
device simple_device;
import "io-memory.dml";

extern int fib(int x);

bank b {
    parameter function = 0;
    register r0 size 4 @0x0000 {
        parameter allocate = false;
        method write(val) {
            log "info": "Fibonacci(%d) = %d.", val, fib(val);
        }
    }
}

header %{
int fib(int x);
%}

footer %{
int fib(int x) {
    if (x < 2) return 1;
    else return fib(x-1) + fib(x-2);
}
%}
```

Writing to the (pseudo-) register `r0` has the effect of printing a log message with the computed Fibonacci number:

```
simics> phys_mem.set 0x1000 6
[dev1 info] Fibonacci(6) = 13.
```

Notable points are:

- The C identifier `fib` is declared in DML using a top-level `extern` declaration.
- The section `footer %{ ... %}` is appended as it is to the C program generated by the DML compiler (`dmlc`). Its contents are not examined in any way by `dmlc`.
- Similarly, the `header %{ ... %}` section is included verbatim at the beginning of the program generated by `dmlc`.
- It is normally better to place C functions in separately compiled files, if possible, and link with the generated DML code. (To do this, you need to modify the Makefile for the module.)
- The function declaration must be included in the header section as well as being declared `extern` in DML. They are usually identical, but not always (in DML 1.0), and thus the C declaration is not automatically generated by `dmlc`; this could change in future releases.
- Header and footer sections could become deprecated or removed in future versions of DML.

The Simics API (a set of C functions and data types) is defined in DML in the library file `simics-api.dml`, which is automatically included by `dmlc`. The corresponding C declarations are defined in header files which are also automatically included by the generated code for a device. Hence, the DML programmer has direct access to the entire Simics API.

For example, this is how the method ***set_read_value()*** is implemented in the standard library. It updates a `generic_transaction_t` structure (a "memop") at the end of a read access to a memory bank has finished, depending on the endianism of the bank:

```
method set_read_value(generic_transaction_t *memop,
                      uint64 value) default
{
    if (!defined $byte_order)
        error
    else if ($byte_order == "little-endian")
        SIM_set_mem_op_value_le(memop, value);
    else
        SIM_set_mem_op_value_be(memop, value);
}
```

Chapter 4

Events and Time

In a hardware simulation, it can often be useful to let something happen only after a certain amount of (simulated) time. This can be done in Simics by "posting an event", which means that a callback function is placed in a queue, to be executed later on in the simulation. The amount of simulated time before the event is triggered is usually specified in a number of seconds (as a floating-point number), but other units are possible; see the *DML Reference Manual* for more information.

We can modify our example to post an event when the register is written to, as follows:

```
dml 1.0;
device simple_device;
import "io-memory.dml";

data int32 delay;
bank b {
    parameter function = 0;
    register r0 size 4 @0x0000 {
        method write(val) {
            $this = 0;
            $delay = val;
            inline $ev.post($delay, NULL);
            log "info": "Posted tick event";
        }
        event ev {
            method event(void *data) {
                ++$r0;
                log "info": "Tick: %d.", $r0;
                inline $this.post($delay, NULL);
            }
        }
    }
}
```

We use the register itself as a counter, which is reset to zero upon a write access; the written value is used as the delay in seconds. Once the event happens, it re-posts itself after the same interval. Note the use of a `data` field to store the delay internally; we could also have chosen to put the counter in the data field instead, or in a more realistic example we could have modeled the counter as an additional register. Also, the ***post*** method must be called with the `inline` keyword, because its argument can be of varying types.

After recompiling and restarting Simics, just enter the command **continue** (or **c** for short). This simply runs the simulation of the hardware. You should see no messages, since there is nothing exciting going on in the machine, except that the clock is ticking away. Press `Ctrl-C` to pause the simulation and get back to the prompt.

Now write a large value to the register:

```
simics> phys_mem.set 0x1000 10000
```

(ignore any error messages about "no associated queue") and enter **c** again. You should see "Tick"-messages being written at fairly short intervals. Press `Ctrl-C` and write a lower value to the register:

```
simics> phys_mem.set 0x1000 1
```

then start the simulation again. The messages are now printed at high speed (although not ten thousand times as fast). The lesson from this is that simulated time is not strictly proportional to real time, and if a machine has very little to do, even 10,000 seconds can be simulated in a very short time.

For simple cases of posting a callback after a specific time, you don't even need to go through the trouble of declaring an event object and its ***event*** method, and then call ***post***. The following DML statement creates and posts an anonymous event which calls a method after a given time `t` in seconds:

```
after (t) call $my_callback;
```

Note that just like calling ***post***, the effect is that the callback is placed on a queue, and the DML program continues immediately with the next statement following `after`, i.e., events are executed *asynchronously*.

Chapter 5

Interfaces and Objects

In addition to attributes, Simics configuration objects can have *interfaces*, which are named groups of methods. In fact, an attribute can be viewed as a very simple interface with two methods (***get*** and ***set***).

A Simics configuration consists of a number of interacting configuration objects; for example, the machine `vacuum` consists of a clock, a memory space, and a RAM. (The clock acts as a pseudo-CPU and is needed to drive the execution.) Each of these is represented as a separate configuration object, and the interaction between the objects is done through interfaces.

To take a concrete example: when a CPU wants to make a memory access, it will call the ***access*** method that is part of the `memory_space` interface implemented by the memory space object. If the memory space then finds some object (e.g., our device) mapped at the accessed address, it will call the ***operation*** method in the `io_memory` interface implemented by the mapped object. (If the mapped object was our DML device, and we had included the standard library file `io-memory.dml`, the device will automatically contain an implementation of the `io_memory` interface, which uses the function number of the mapping to pass on the operation to the ***access*** method of the correct register bank in the DML program.)

A very basic question in this context is "how does one object find another object?" Usually, objects are connected through attributes; e.g., a CPU object could have an attribute that holds a reference to a memory space object, and the memory space object has an attribute that contains mapping information (which includes references to the mapped objects), and so on. Such bindings are typically set up in the configuration scripts for a simulated machine, and are not changed after the initialization is done.

DML has built-in support both for letting your device implement any number of interfaces, and for connecting other objects to your device that implement particular interfaces.

5.1 Implementing an Interface

The following is a naive implementation of the `io_memory` interface, designed to be used with the example device in Section 2.1 (note the references to `$b` in the code). To try it, simply remove the line "`import "io-memory.dml";`" from the original program, and instead add the below code (before or after the `bank b {...}` section).

```
implement io_memory {
    method map(addr_space_t mem, map_info_t info)
           -> (int status)
    {
        log "info": "Mapped function %d at %#x",
                    info.function, info.base;
        status = 0;
    }
    method operation(generic_transaction_t *memop,
                     map_info_t info)
                -> (exception_type_t ex)
    {
        if (info.function == $b.function) {
            local uint32 offset;
            offset = (memop->physical_address
                      - info.base
                      + info.start);
            log "info": "Accessing internal address %d",
                        offset;
            call $b.access(memop, offset, memop->size);
            ex = Sim_PE_No_Exception;
        } else {
            ex = Sim_PE_IO_Not_Taken;
        }
    }
}
```

(The ***map*** method does not have to do anything, but should return zero. The ***operation*** method should return an "exception code" which is `Sim_PE_No_Exception` if all went well.)

The method declarations within an `implement` section are translated directly into C functions as expected by the Simics API; a pointer to the device object itself is automatically added as the first argument to each function. The methods can only have one or zero output parameters, which correspond directly to return values in C. In this example, the C function signatures for ***map*** and ***operation*** are:

```
                  int map(conf_object_t *obj,
                          addr_space_t mem,
                          map_info_t info);

exception_type_t operation(conf_object_t *obj,
                           generic_transaction_t *memop,
                           map_info_t info)
```

Try recompiling and testing the device with this implementation of `io_memory`. For further details about the `io_memory` interface, and other Simics interfaces and datatypes, see the *Simics Programming Guide* and the *Simics Reference Manual*. (In general, the DML built-in constructs and standard libraries try to hide as much as possible about the details of the Simics API from the user.)

5.2 Connecting Objects

As mentioned previously in this chapter, the usual way to make one object accessible from another object, so that the latter (the caller) may communicate with the former (the callee) via the interfaces it implements, is to let the caller have an attribute that can hold an object reference, and set that attribute to point to the callee object (typically done in an initialization script). Although it is possible to write an `attribute` definition by hand, suitable for connecting an object with a particular interface, it is much better to use a `connect` definition, which creates such an attribute with minimal effort.

An interface, in Simics, is a struct containing function pointers, and the definition of the struct must be visible both to the caller and the callee. (The standard Simics API interface definitions are included automatically by the DML compiler.) The convention in the Simics API is to use a C `typedef` to name the struct type, using the suffix `_interface_t`, and the DML compiler by default follows this convention when it generates interface-related code. E.g., the `io_memory` interface is described by a datatype `io_memory_interface_t`, which is a struct containing two function pointers ***map*** and ***operation***. If the user wants to create new interfaces, he must write his own struct definitions; this is demonstrated below.

5.2.1 Connect Example

In the following example, we will create a second device and connect it to the first device via a user-defined interface. We start with the example device in section 2.1, and add the following declaration:

```
connect plugin {
    interface talk {
        parameter required = true;
    }
}
```

Then, we replace the line "`log "info": ...;`" with the following C function call:

```
$plugin.talk.hello($plugin);
```

The members of an interface are C functions, not DML methods, so do not use the `call` or `inline` keywords here. Also note that `$plugin` itself is given as the first argument to the function. This is standard in most interfaces used in Simics.

Our device will now have an attribute named *plugin*, which can hold object references; the attached objects are required to implement the `talk` interface. However, we cannot yet compile the module, since it is missing the definition of the interface.

5.2.2 Sharing Definitions

There are two main kinds of source files that may be shared by modules:

- DML files to be imported
- C header files to be included by the generated sources

We will put the definition of our simple `talk` interface in a shared DML file. First of all, we need a place to put the file. Create a new directory `[home]/workspace/include` (e.g., using `mkdir` on a Unix system), and open the new file `[home]/workspace/include/talk.dml` in your editor. Put the following text in the file and save it:

```
dml 1.0;

struct talk_interface_t {
    void (*hello)(conf_object_t *obj);
}
```

(such a `struct` declaration in DML will both inform the DML compiler about the existence of the struct type, as well as cause a corresponding `typedef`-declaration in the generated C code).

Then, add this line to the example device code, to import the new file:

```
import "talk.dml";
```

Finally, edit the `Makefile` for the example device: `[home]/workspace/modules/simple_device/Makefile`, and add the following option to the definition of `DMLC_FLAGS`:

```
-I$(SIMICS_WORKSPACE)/include
```

(telling `dmlc` where to look for additional include files).

You should now be able to compile the example device with the `connect` added as described above.

Sharing C header files (when necessary) is similar to the above: just add a C compiler "`-I...`" flag to the makefile, and instead of the DML `import` directive, use a C `#include` within a `header` section, as in:

```
header %{
  #include "stuff.h"
%}
```

5.2.3 A Plug-in Module

To complete the example, we need an object that speaks the `talk` interface, which we can use to connect to our device. For this purpose, we add a new module to our workspace, as follows (cf. Section 2.1):

```
$ [simics]/bin/workspace-setup --force --dml=plugin_module \
    [home]/workspace
```

(note that the `--force` flag is needed to add a module to an existing workspace).

Edit the generated skeleton file `[home]/workspace/modules/plugin_module/plugin_module.dml` to look like this:

```
dml 1.0;
device plugin_module;

import "talk.dml";

implement talk {
    method hello {
        log "info": "Hi there!";
    }
}
```

The only way to use the objects of this class is through the `talk` interface - there are no memory-mapped registers or similar connections. (Do not take the term "device" too literally; a DML source file does not have to model a piece of hardware - it just defines a component class that can be loaded in Simics.)

Also edit the device makefile: `[home]/workspace/modules/plugin_module/Makefile`, and add the option `-I$(SIMICS_WORKSPACE)/include` to the definition of `DMLC_FLAGS`, just like we did previously for the first example device.

5.2.4 Testing the Connect

Simply running `make` (or `gmake`) from the `[home]/workspace` directory should now compile both modules. Do this and restart Simics as before. (We assume that you have by now added the commands from Section 2.2 to the startup script, as described in Section 2.3, so that you do not have to enter them manually each time you run Simics.)

Enter the following command in the Simics console:

```
simics> @SIM_create_object("plugin_module", "dev2", [])
```

to create an instance of the plugin device. (Simics loads the module automatically.)

Then, set the *plugin* attribute of the first example device to point to the new object:

```
simics> dev1->plugin = dev2
```

We are now ready to run. Make a memory access to the first device:

```
simics> phys_mem.get 0x1000
```

and you should see the following output:

```
[dev2 info] Hi there!
42
```

Virtutech, Inc.

1740 Technology Dr., suite 460
San Jose, CA 95110
USA

Phone +1 408-392-9150
Fax +1 408-608-0430

http://www.virtutech.com