

SIMICS
MODEL BUILDER

**VIRTUTECH DML –
DEVICE MODELING
LANGUAGE**

MAY 2009

JAKOB ENGBLOM

WWW.VIRTUTECH.COM

INTRODUCTION

Virtualized Systems Development[™] is a development methodology where the actual hardware of a system is replaced with a virtual model running on a workstation or PC. The virtual hardware can run the same binary software as the physical hardware, fast enough to be used as an alternative to physical hardware for software development. The virtual hardware provides additional benefits like better debugging facilities, checkpointing and restart at any point, superior convenience and stability, access to the target long before prototype hardware to start software development early, and the ability to test faults and boundary cases with complete control and precision.

The key to realizing the benefits of virtualized software development for a particular application is the ability to quickly develop high-performance virtual systems. This is the task of *system modeling*, and this Virtutech white paper will discuss how system modeling is supported and improved by the Virtutech-developed device modeling language known as DML.

At core, Simics is an extremely fast transaction-level model (TLM) simulator. Simics features an efficient simulation infrastructure that has been honed by active use for more than ten years, very fast processor simulators, optimized target memory handling, and a proven API for device modeling¹. All Simics models are transaction-level, in all their interfaces.

DML is a language to quickly create fast functional models of hardware devices, created explicitly to address the productivity issue for transaction-level virtual platforms.

WHY DML?

Creating transaction-level virtual platform models is fundamentally a programming task. Today, most of this work is performed in general-purpose programming languages like C, C++, Python, Java, and SystemC. These languages do not provide constructs that really correspond to the concepts involved in creating transaction-level models of hardware. Features that are

¹ For more on Simics performance, see the [white paper on Simics speed](#) on the Virtutech website.

missing and have to be created by each programmer are for example memory-map decoding, bit-field manipulation, interpretation of network packets and other packed structures, access to target memory, and working with endianness and word-length different from the host. In the end, each simulation model tends to contain a lot of logic repeated from other models, but which is hard to break out into a library since it is intertwined with custom code for each model.

The lack of explicit language support for the TLM domain leads to suboptimal programmer productivity and carries a large risk of functional mistakes and performance-degrading modeling mistakes.

Further hampering productivity, a large part of the model code in current simulators is interfacing code towards the simulation framework itself. Such code is voluminous and time-consuming to write but add no real value to the final models. The result is code where it is very easy to make mistakes, both functional and performance-wise, and that takes a significant amount of time to write.

Virtutech has been building TLM virtual platforms since 1991, and realized the importance of the model programming problem a long time ago. In 2004 we started to use a language called DML, Device Modeling Language, for internal development of models. In 2005, DML was released to Virtutech customers. It has since undergone two major revisions and is currently at version 1.2. It has proven a real boon to modeling productivity for all users, both internal and external to Virtutech.

WHEN DML?

A system model in Simics consists of four broad classes of hardware models:

- Processor cores – the CPUs actually running processors instructions. For example PPC 464, MPC e500, Core 2 Duo, MIPS 5Kc, or ARM9.
- Interconnects – networks and buses connecting devices, machines, boards, and cabinets together. For example serial, Ethernet, I2C, PCI, SCSI, USB, or MIL-STD-1553.
- Memory – RAM, ROM, EEPROM, FLASH, and other types of memory devices that store large amounts of code or data.

- Devices – anything else. All the peripheral units that move data between machines or do work that is not instruction processing. Timers, interrupt controllers, ADC, DAC, network interfaces, I2C controllers, serial ports, LED drivers, displays, media accelerators, pattern matches, table lookup engines, and memory controllers are just some examples of devices.

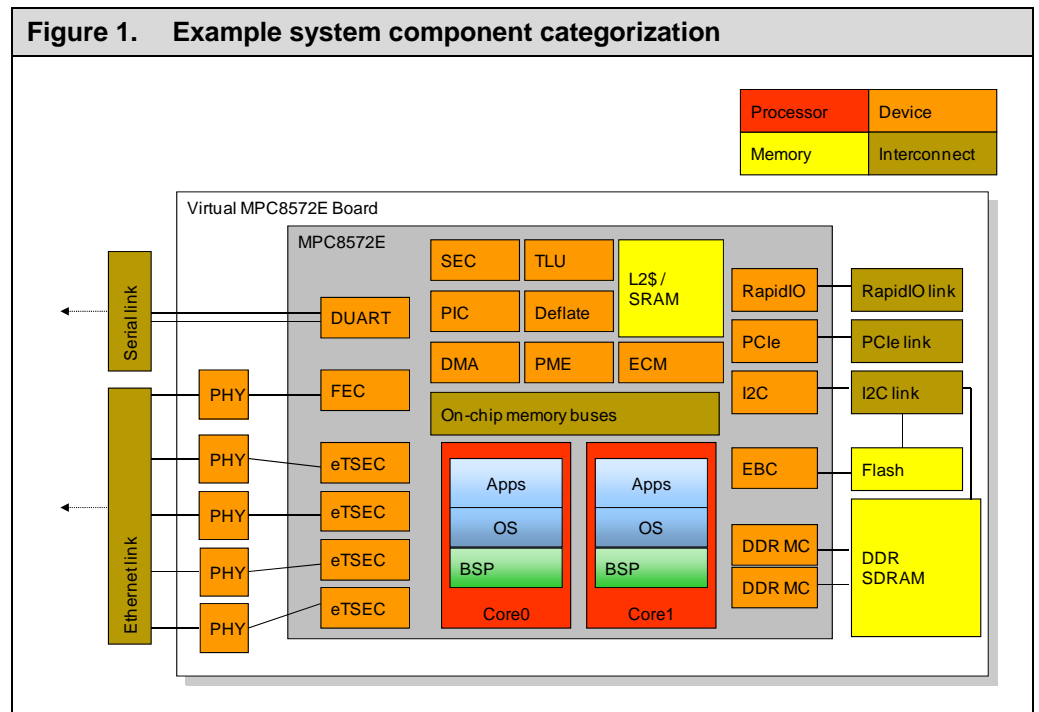


Figure 1 shows an example decomposition of a system into these categories.

In general, most of the work in creating a new system model is spent modeling devices, and they are the most numerous and least standardized of the hardware components. DML is designed to help in this task, as device modeling is the task that experience tells us could benefit the most from a better programming system.

Processors and memory models are highly reusable as the number of varieties is comparatively low. They also need significant detail work in order to ensure simulation performance. In Simics, processor models are either provided by Virtutech, or written in any language and plugged into Simics using the Simics processor API (available in Simics 4.0 and later versions).

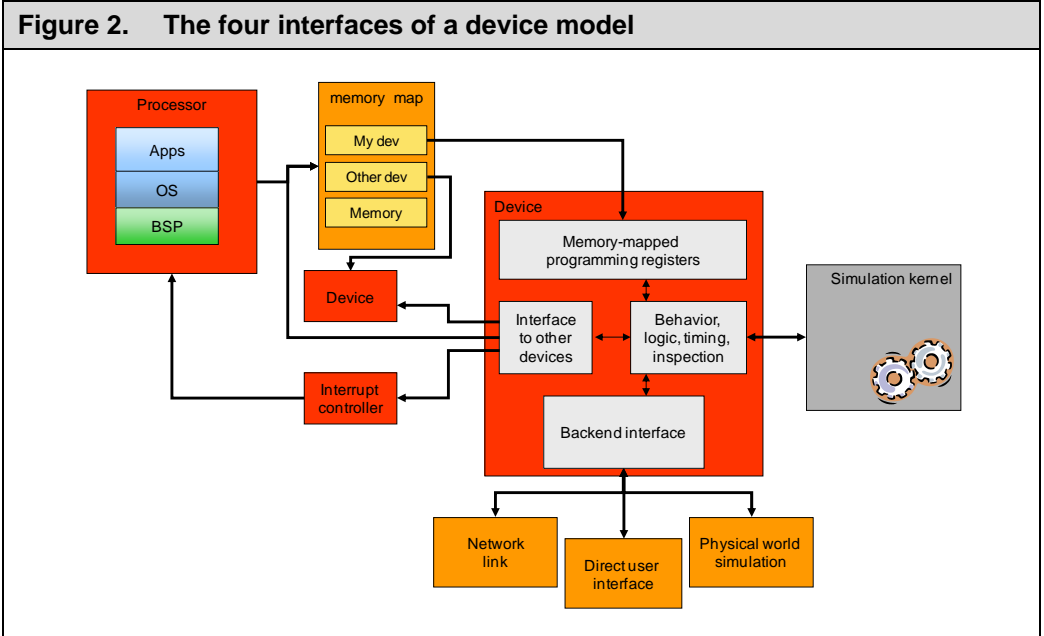
Interconnects are usually standardized and reusable across simulated systems. They tend to be quite complex when implemented to support distributed and multithreaded simulation, and the implementation of one interconnect can be quite different from the other. For this reason, interconnects in Simics are usually programmed in C or C++ since there is less commonality to be leveraged into a custom language.

Thus, devices form the bulk of the modeling effort, and is repetitive enough to warrant creating a domain-specific language to increase productivity and simplify programming.

DEVICE MODELING IN SIMICS

A hardware device model essentially has four interfaces, as illustrated in Figure 2.

- Programming register map.
- Communication networks and links.
- Tightly-coupled devices.
- Simulator core.



The memory-mapped programming register map is where the device driver software writes and reads registers in the device to control the behavior of the device. This is often known as the “front end” of a device model. It can be as simple as a few bytes or contain many thousands of registers with varying size.

Communication buses and networks are where the device model communicates with the external world (external to the chip it is contained in). Typical examples are serial lines and Ethernet networks, but it could also be something as simple as I2C or complex as a model of a plant being controlled. In Simics, most such interfaces are implemented with an explicit network or communications link. This makes it easier to support multithreaded simulation, and also ensures that all input and output to a device model can be recorded and replayed.

Device can also be *tightly coupled* to each other. Typically, devices within the same chip need to access specific function of other devices that are not really suitable to model as explicit communications networks. For example, a multiprocessor system controller will need to route interrupts from devices to processors and pass interrupts between processors in the system. Network processing accelerators could have direct data connections to the network

interfaces of an SoC where network frames go directly without touching system memory. Devices also often access target memory to do DMA operations and grab descriptor tables for operations.

The *interface to the simulator core* is used to drive time forward and to perform housekeeping and infrastructure tasks. In Simics, this includes checkpointing, support for attributes, reverse execution, logging, posting and reacting to events and haps, and general access to the external world. The part of the model that interacts with the simulator core is also the part that ties the activity on the other interfaces together. Note that a large portion of the kernel interface code is automatically generated for DML models.

If you want to know more about the general principles of device modeling in Simics and the applicable methodology, please see our white paper on modeling. This white paper is focused on the specifics of the DML language.

INTRODUCING DML

DML is a textual language that provides a way to write transaction-level device models with less code and fewer mistakes than plain C/C++ and a simulator API. DML models are much more concise than corresponding C language models, and much easier to write, read, understand, and maintain. DML lets users easily express complex register maps for devices, including bit fields and sparse register maps.

DML also explicitly supports an iterative development style for device models by providing default implementations for many device aspects and strong support for marking unimplemented aspects. DML supports and encourages inline documentation in a model, and documentation is extracted and used to help Simics provide a better end-user interface for a model. Eclipse and Emacs editing modes specific to DML are provided with Simics to provide a custom smart editing environment.

Compilation Process

DML does not compile directly to binary models, rather DML is compiled to C code that is then compiled by the native C compiler. The DML compiler also generates the plumbing code needed to connect a particular model to Simics, reducing the Simics API knowledge needed to create models and keeping the code relatively free from explicit Simics API calls. This includes

automatic support for checkpointing, attributes, reverse execution, logging, and other Simics features.

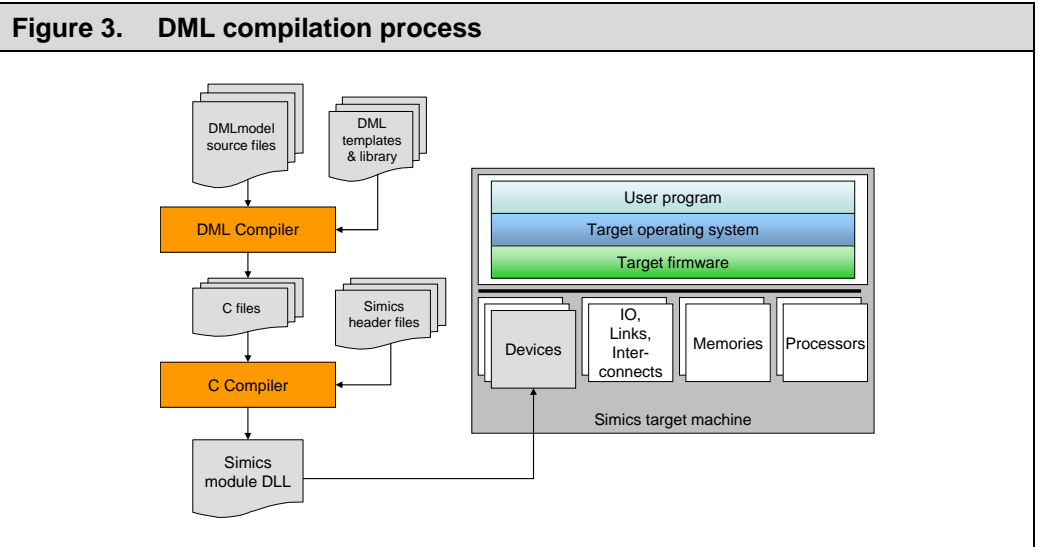


Figure 3 illustrates the steps of the DML compilation process. Note that the entire process is transparent to DML users, who just need to invoke the Simics module `makefile` automatically generated by Simics to compile a DML module. The C code is compiled like a Simics module written in C, using the same Simics API header files as all other Simics modules.

The common denominator for all Simics modules is the C-level Simics API. By generating C code with Simics API calls, DML code integrates naturally with any other Simics module. It also makes it possible to combine C and DML source code into a single module, which can be used to wrap existing C-language simulations behind a DML frontend, or make use of particular C-language libraries to implement module functionality.

Reactive TLM

DML is designed to code models in a reactive style. DML describes models as a set of methods (functions) which are called from the simulation kernel and across the device interfaces whenever something occurs that the model has to respond to. There is no main loop in a DML model, the simulation kernel and DML compiler takes care of sequencing and activation of the different pieces of sequential code in a model.

This is the industry-standard way to write high-performance transaction-level models. In SystemC terms, it is equivalent to using `SC_METHOD` rather than `SC_THREAD`. In C, it means creating a model to be driven by function calls from other parts of the simulation rather than running multiple operating-system threads and using inter-thread communication and coordination.

The benefit of a reactive model is first of all higher performance, since there is no thread-level context-switching cost incurred when activating models. Creating a reactive model also makes it easier to stop the simulation at any time and investigating and checkpointing its state, as no state is hidden on a C-level call stack. It is also necessary in order to support checkpointing and reverse execution, since it provides the simulation kernel with full control over the model state.

Model Performance

Not all models are equal, and the final performance of the simulation is directly affected by how models are designed and written. Modelers need to think carefully about the design of a model to make it fast, and make sure that no implementation details accidentally spoil performance.

With DML, it is harder to make performance mistakes. The DML language itself encourages a reactive TLM style of coding that does not rely on periodic updates to implement functionality. The DML compiler generates the integration code to the simulation platform, which removes the risk of misusing the platform API in a performance-zapping way. DML itself does not introduce any overhead compared to native C coding of a model.

The DML compiler can also optimize the generated code and do analysis on a device model to do smart things to the final code. For example, the implementation of a device register map address decoder can be generated in a way suitable to how a particular register map is setup.

DML FEATURES

Register Maps

The most obvious contribution of DML is in simplifying the specification and implementation of device programming registers. This was the main problem that we wanted to solve when DML was initially designed. The typical way a

device register map decoder is written in a C-family language is a “big switch”.

Figure 4. UART programming register decoder in SystemC, “big switch”

```
int uart::IPmodel(accessHandle t)
{
    // get the data from transaction
    data.set(t->getMData());
    // READ behaviour
    if (t->getMcmd()==Generic_MCMD_RD){
        // Which offset was accessed?
        switch ((unsigned int)t->getMAddr()){
            case 0x0:
                // address 0x00 has different behavior based on DLAB
                if (DLAB)
                    (*(gs_uint8*)data.getPointer())=DLL;
                else{
                    if(FCR0){
                        RBR=RCVR_fifo.read(); // Read from FIFO if enabled
                        if(RCVR_fifo.num_available()<rcvr_trigger_level)
                            IssueInterrupt(CRDA);
                    }
                    if( !FCR0 || RCVR_fifo.num_available()==0 )
                        LSR=LSR&0xFE; // Clean DR (Data Ready) bit
                    (*(gs_uint8*)data.getPointer())=RBR;
                }
                break;
            case 0x1:
```

Figure 4 shows an excerpt from a typical big switch code. The code needs to work out if a memory access is a read or write, and then look at the address and take the appropriate action. The result is normally two large switch statements, one for the read and one for the write case. The drawback of this coding style is that it mixes the declaration of the register map with the implementation of the behavior for each register, and that read and write behavior tends to end up in two separate locations.

Note that doing the outer switch on the address accessed and then for each case determining if the operation is a read or write ends up being just as hard to read. The core problem here is that doing this type of dispatch is not part of the C-style languages, and you end up coding a particular implementation of the decoder rather than simply declaring it.

In DML, the declaration of the register map is much more declarative in style. As shown in Figure 5, the model source code can describe the register layout

(which registers have which addresses) separately from the implementation, making it much easier to read.

Figure 5. UART programming register layout in DML

```
bank uart {
  parameter register_size = 1;
  parameter byte_order = "little-endian";

  register rbr @ 0x0 "Receiver Buffer register";
  // non-mapped registers to take care of multiple personality
  // of register at offset 0
  register thr @ undefined "Transmitter Holding Register";
  register dll @ undefined "Divisor Latch LSB";

  register ier @ 0x1 "Interrupt Enable register";
  // non-mapped register to take care of the multiple personality of
  // register at offset 1
  register dlm @ undefined "Divisor Latch MSB";

  register iir @ 0x2 "Interrupt Identification register";
  // non-mapped register to take care of multiple personality of
  // register at offset 2
  register fcr @ undefined "FIFO Control Register";

  register lcr @ 0x3 "Line Control register";
  register mcr @ 0x4 is (unimplemented) "MODEM Control register";
  register lsr @ 0x5 "Line Status register";
  register msr @ 0x6 is (unimplemented) "MODEM status register";
  register scr @ 0x7 "Scratch pad register";
}
```

The actions to be taken on memory accesses are described in separate read and write methods for each register, and these are usually defined in a block of code separate from the main register map declaration. Figure 8 shows an example of how this code looks. You are allowed to put both the declaration of the register sizes and offsets and the definition of the actual functionality into the same block of code, but it is strongly recommended that you separate them, to make the source code easier to read. Note that there is no need to repeat the size or offset of a register when declaring the functionality, the DML compiler takes care of combining all information specified for a register.

Note the use of parameter statements in Figure 5 to specify the endianness and default register size for the register bank called uart. Such parameters are common throughout DML, providing values for defaults or specifying the particular behavior of a particular object. Figure 7 also shows some

parameters giving the reset values of a register, which is another way in which DML makes device coding easier.

Bit Field Decoding

Device programming registers tend to be divided up into fields consisting of a few bits each, and decoding the contents of such fields ends up as a series of very error-prone bit masking and shifting operations in most C code. Figure 6 shows another SystemC example of such code, looking at data which is at the lowest two bits of a register.

Figure 6. Bit field decode in SystemC

```
case 0x3:
    LCR=*(gs_uint8*)(data.getPointer());
    // This is the word size, stored on a convenience
    // attribute for easy AND
    if      ((LCR&0x03)==0) mask=0x1F;
    else if ((LCR&0x03)==1) mask=0x3F;
    else if ((LCR&0x03)==2) mask=0x7F;
    else if ((LCR&0x03)==3) mask=0xFF;
    break;
```

As shown in Figure 7, DML allows the user to declare the bits inside each register that make up each field. It is possible to use both little-endian and big-endian (Power Architecture-style) bit field numbering, to align with how the device manuals describe the bit fields.

Figure 7. Bit field decode in DML

```
register lcr {
  parameter soft_reset_value = 0x00;
  parameter hard_reset_value = 0x00;

  field wls          [1:0] "Word length select ";
  field stb          [2]  "Number of stop bits (0 = 1, 1 = 2)";
  field pen          [3]  "Parity enable (0 = disable, 1 = enable)";
  field eps          [4]  "Even parity select (0=Odd, 1=Even)";
  field stick_parity [5]  "Stick parity";
  field set_break    [6]  "Set break";
  field dlab         [7]  "Divisor latch access bit";

  // After a write to this register, check the contents of WLS and
  // set the character length mask appropriately
  method after_write (memop) {
    if      ($wls == 0) $mask = 0x1F;
    else if ($wls == 1) $mask = 0x3F;
    else if ($wls == 2) $mask = 0x7F;
    else                $mask = 0xFF;
  }
}
```

The last bit of code in Figure 7 performs the same work as the code above, and note that no explicit masks are needed. The DML compiler takes care of generating the shifting and masking code, and the user does not need to care about how this is done.

Endianness and Partial Accesses

An advantage of using a declarative style for register maps is that the DML compiler knows the register layout explicitly. This enables the DML compiler to automatically generate code supporting tricky cases like accesses that hit only part of a register (a single byte in a 16-bit register, for example), and accesses that cover several registers in a single memory operation.

Depending on the specification of the target device, such accesses are either flagged as software errors or handled by routing accesses to the correct parts of a device. This is set by a simple parameter in the source code for a bank.

Endianness is also explicitly declared for a register bank, and that endianness is handled correctly regardless of the endianness of the host. Note that there is both byte endianness, which determines how data arriving in a memory transaction is interpreted, and bit endianness, which deals with how bits are numbered inside a register. These two are independent.

DML makes handling cases like a two-byte access in the middle of an eight-byte big-endian register on a little-endian host trivial. Coding such an access in C is fraught with problems and risks for error. It also reduces the mental strain on the programmer who has to deal with mixed endianness in a target system.

C-like Core Language

The code that actually performs the work in DML is an extended subset of C. The code can access fields and variables defined in the model using a \$-prefix to names, and define and call methods. The DML compiler ties the local definitions of behavior together into a coherent device model, the programmer needs not care about sequencing. Small snippets of core code are shown in Figure 7 and Figure 8.

Figure 8. UART register access handling in DML

```
bank uart {
  register rbr {
    method read -> (uint8 chr) {
      if ($fcr.fifo_enable == 1) {
        // RBR is filled up here from the rcv FIFO
        call $rx_fifo.dequeue;
      }
      chr = ($this & $char_len_mask);
      $uart.lsr.dr = 0;
      call $update_interrupt;
    }
    method write(value) {
      // pass write along to thr register
      // since rbr and thr are at the same offset
      call $thr.write(value);
    }
  }
  ...
}
```

Compared to normal C, some additional features are available in DML. In particular, C++-style new and delete is used to manage dynamic memory, and there is support for try-catch exception handling. Other extensions help express common simulation functions like logging information, error reporting, and asserts concisely.

The use of C style is intentional to make DML easier to learn. As a first approximation, DML can be considered as a macro language tying together snippets of C code. There is however much more to DML than that.

Templates

To reduce code size, DML uses templates to describe recurring functionality, for example, “always zero”, or “clear on read”. Users can define their own templates. Groups and arrays are provided in DML to describe repeating patterns and register structures. Using DML mechanisms, even very large and complex register banks can be described succinctly. We have many examples of devices with several thousand registers modeled in DML. Thanks to the domain-specific nature of DML, these mechanisms are more powerful and easier to use than C++ templates.

The DML compiler comes with a set of predefined templates for common register cases. A single register or field can combine several templates, as long as the templates do not affect the same aspect of the register behavior. The table in Figure 9 below gives some examples of DML templates available in Simics 4.2 with DML 1.2.

Figure 9. Example templates for common register and field behaviors	
<code>clear_on_read</code>	When read, return the current value and set the value to zero.
<code>ignore</code>	Writes have no effect, reads always return zero
<code>read_constant</code>	All reads return the value set in the model source code, writes have no effect
<code>reserved</code>	Log accesses as reading or writing reserved bits, remember written values, read last value written.
<code>read_only</code>	Writes are ignored and logged
<code>unimplemented</code>	Log accesses as accessing unimplemented feature to Simics console, remember written values, read last value written.
<code>write_1_clears</code>	Writes clear the bits marked by ones in the written value.

Templates usually carry parameters that specify aspects of their behavior, making them quite general in applicability. For example, the constant template carries a value parameter providing the value to which it is fixed.

It is worth noting that register semantics in Simics are often expressed in a combination of computation of the results corresponding to the target machine computations and simulation-specific side effects like logging. This makes it easier to support iterative development of models, and provides a richer communication of the information the simulator provides about the system and its execution.

Arrays and Groups

Hardware often contains repeated groups of registers with identical functionality repeated multiple times. It can be the per-processor registers in a multiprocessor interrupt controller, or replicated functional units in an accelerator. To make the specification of such units simpler, DML contains the concepts of arrays and groups. Groups group related registers together, and arrays can repeat individual registers or groups of registers multiple times.

Figure 10. DML groups and arrays used in an interrupt controller

```

bank pic {
  ...
  group GT[g in 0 .. NUM_TIMER_GROUPS - 1] {
    register TFRR is (read_write)
      "Timer frequency reporting register";
    register GTCCR[NUM_TIMER_INTERRUPTS] is (gt_curcount)
      "Global timer i current count register";
    register GTBCR[NUM_TIMER_INTERRUPTS] is (gt_count)
      "Global timer i base count register";
    register GTVPR[NUM_TIMER_INTERRUPTS] is (VPR)
      "Global timer i vector/priority register";
    register GTDR[NUM_TIMER_INTERRUPTS] is (DR)
      "Global timer i destination register";
    register TCR is (gt_control)
      "Timer control register";
  }
  ...
}

```

Figure 10 shows an example of the use of arrays of groups from an interrupt controller. Note that you have a number of sets of GT registers, and each such set contains several register arrays like GTCCR. The declaration also makes extensive use of DML compile-time constants so that the exact number of registers can be set from a file including this general declaration. In this way, the same source code can be shared across multiple similar devices. The is statements also indicate the use of templates for functionality, some of which are user-defined (MSIR, VPR, DR), and some which are standard with Simics (read_only).

Putting such nested structures into the right places in memory would be painful if the offset of each register would have to be specified individually. To support this, DML features computed offsets using index variables, as shown in Figure 11.

Figure 11. DML computed offsets

```
bank pic {  
    ...  
    group GT[g in 0 .. NUM_TIMER_GROUPS - 1] {  
        register TFRR @ 0x010f0 + 0x1000*$g;  
        register GTCCR[NUM_TIMER_INTERRUPTS] @ 0x01100 + 0x1000*$g + 0x40*$i;  
        register GTBCR[NUM_TIMER_INTERRUPTS] @ 0x01110 + 0x1000*$g + 0x40*$i;  
        register GTVPR[NUM_TIMER_INTERRUPTS] @ 0x01120 + 0x1000*$g + 0x40*$i;  
        register GTDR[NUM_TIMER_INTERRUPTS] @ 0x01130 + 0x1000*$g + 0x40*$i;  
        register TCR @ 0x01300 + 0x1000*$g;  
    }  
    ...  
}
```

Memory Layouts

Hardware devices quite often operate on data structures in main memory, without involvement of the main processor in a system. Typical cases are descriptors, structures describing a set of work for a device to do, and data network packets that are processed (or deposited in memory) by devices. To support the modeling of such devices, DML has a memory layout data type. Memory layouts look similar to structure types in C, and are used as types for variables. Unlike C structs, DML layouts explicitly map directly to the data layout in memory (in C, the compiler is actually free to insert padding between fields).

Figure 12 shows a snippet of DML code involving a layout. Note that the layout has an explicit endianness, and includes single bits in bit fields as addressable units. It is used as the data type for a local variable, into which the bytes from a block of memory are copied. The manipulation of a layout is local to the code of the device, which avoids repeated calls to the simulated memory system to collect the data, enhancing locality and speed in the simulator.

Figure 12. Memory layouts in DML

```
// example from Simics documentation
typedef layout "big-endian" {
    uint32 addr;
    uint16 len;
    uint8 offset;
    bitfields 8 {
        uint1 ext @ [0:0];
    } flags;
} sg_list_block_row_t;

...
{
    // local variable of layout type
    local sg_list_block_row_t row;
    memcpy(&row, ptr, sizeof(row));
    ptr += sizeof(row);
    if (row.flags.ext) {
        ...
    }
}
```

Interfaces to Other Models

As discussed above, models also need to interface to other models within the same tightly-coupled system component and to various interconnect links outside. In Simics, all such interfaces are expressed as sets of function calls called interfaces, which a model implements in order to let other models call it over that interface. It is essentially the same as an abstract interface in C++ or Java. The interfaces are unidirectional, providing a way for one device to call into another device. In DML, this is provided by the `connect` and `implement` constructs.

Incoming connections are defined in an `implement`, and define the behavior for each function call in the interface. Outgoing connections in `connect` statements provide the model with a configuration attribute that can be used to tell it which device to connect to. This is set by the system configuration, and provides the model with what is essentially a pointer to the other device. However, that pointer is stored in an attribute, and is thus fully checkpointable and configurable at runtime.

Simics provides common header files and base implementations for interfaces like PCI, RapidIO, Serial, I2C, and Ethernet, to ensure model interoperability and design reuse.

Simics Attributes

A key feature of the Simics simulation infrastructure is the use of *attributes* to make the state of the simulation explicit. All relevant state in a Simics model is stored in attributes, and by reading all attributes from an simulation object you get a complete image of its state that you can later restore. In a C or C++ Simics model, attributes have to be manually registered with the simulation infrastructure. DML makes this process much simpler, by automatically generating attribute creation and support code for all registers and connects in a model.

Sometimes you need to define new model attributes explicitly in DML. This is used to do things that do not really fit into the categories of “registers” or “connects”. For example, supporting device configuration information that results from the platform setup rather than from software activities, like interrupts wired high or low, or the interrupt latency of a device. Other examples include attributes used to disable and enable fault injection in a model, or maintaining various activity counters.

Figure 13. Explicit attribute in DML

```
attribute interrupt_latency {  
    parameter documentation = "interrupt latency (ns)";  
    parameter allocate_type = "int32";  
}
```

Figure 13 shows how explicit attributes are declared in DML. The default behavior is to store a single value and allow this to be get and set from the Simics attribute API, but users can also override the attribute `get()` and `set()` methods to have attributes do anything when accessed.

Figure 14. Coding a Simics attribute in C

```
// Storage for the value attribute contents have to be put in the
// device model internal data representation structure
typedef struct sample_device {
    log_object_t log;
    int value;
} sample_device_t;
...
// get and set functions to convert attribute value to and from
// interoperability format and checkpoints
static set_error_t
set_value_attribute(void *arg, conf_object_t *obj,
                   attr_value_t *val, attr_value_t *idx)
{
    sample_device_t *sample = (sample_device_t *)obj;
    sample->value = val->u.integer;
    return Sim_Set_0k;
}
static attr_value_t
get_value_attribute(void *arg, conf_object_t *obj, attr_value_t *idx)
{
    sample_device_t *sample = (sample_device_t *)obj;
    return SIM_make_attr_integer(sample->value);
}
...
// API call actually creating the attribute
SIM_register_typed_attribute(
    sample_class, "value",
    get_value_attribute, NULL,
    set_value_attribute, NULL,
    Sim_Attr_Optional,
    "i", NULL,
    "The <i>value</i> field.");
```

For comparison, Figure 14 shows the code needed to register an attribute in plain C using the Simics API. It is very repetitive and quite voluminous. Similar things would have to be done in any simulation infrastructure supporting the same features as Simics does, even if the exact way of expressing it might differ. Using explicit API calls to implement functionality is always more expansive in terms of code lines than using implicit generation of the API calls from a higher-level language like DML.

Functionality Templates

Simics also provides ready-made packages of templates and related code to provide a quick start in modeling devices with particular interfaces. DML has been designed to facilitate the provision of such reusable code in a way that makes it easy to extend in arbitrary ways. This makes creating even fairly

complex device models much easier, and reduces the code size since repetitive code is hidden inside the standard template files included with Simics.

Figure 15 shows the minimal PCI device implemented in DML for Simics. The crucial statement is `import "pci-device.dml"`, which imports the basic PCI skeleton from Simics into the device. We then follow the rules for this device template, and fill in the `pci_config` bank.

Figure 15. Sample PCI device in DML

```
dm1 1.2;

device sample_pci_device;
parameter documentation = "This is a very simple PCI device.";

import "io-memory.dml";
import "pci-device.dml";

bank pci_config {

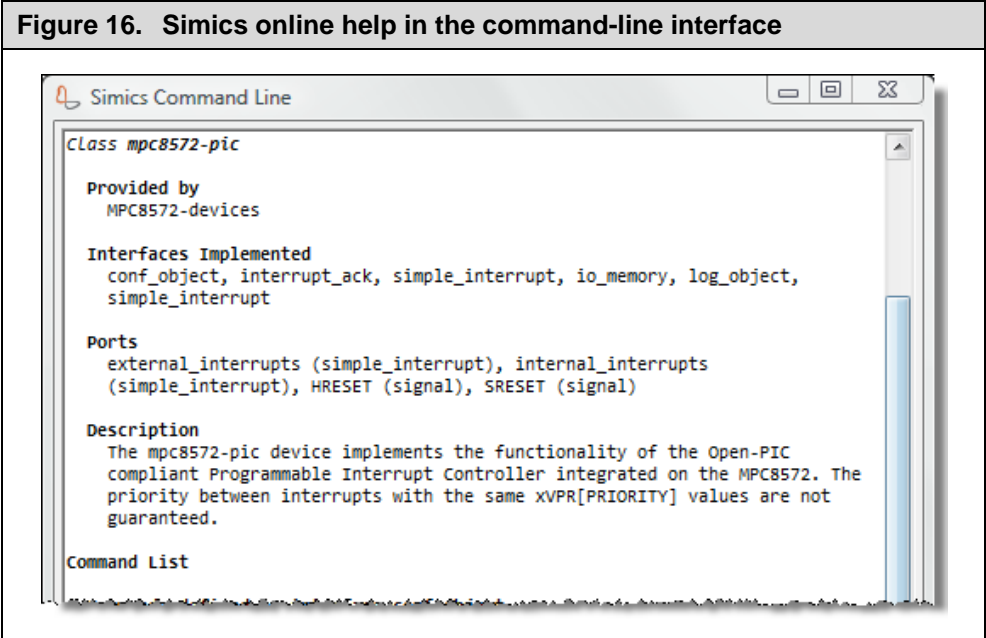
    parameter base_address_registers = ["bar"];

    register vendor_id { parameter hard_reset_value = 0x104C; }
    register device_id { parameter hard_reset_value = 0xAC10; }
    register bar @ 0x10 is (memory_base_address) {
        parameter size_bits = 8;
        parameter map_func = 1;
    }
    register base_address_1 @ 0x14 is (no_base_address);
    register base_address_2 @ 0x18 is (no_base_address);
    register base_address_3 @ 0x1C is (no_base_address);
    register base_address_4 @ 0x20 is (no_base_address);
    register base_address_5 @ 0x24 is (no_base_address);
}

bank regs {
    parameter function = 1;
    // add more registers here
}
```

Inline Documentation

The large amount of documentation contained in a DML file is exposed to the user in several ways. It is used in the GUI device viewer shown in Figure 17, and is also accessible from the Simics online help system from the command-line, as seen in Figure 16.



Integration with C and C++

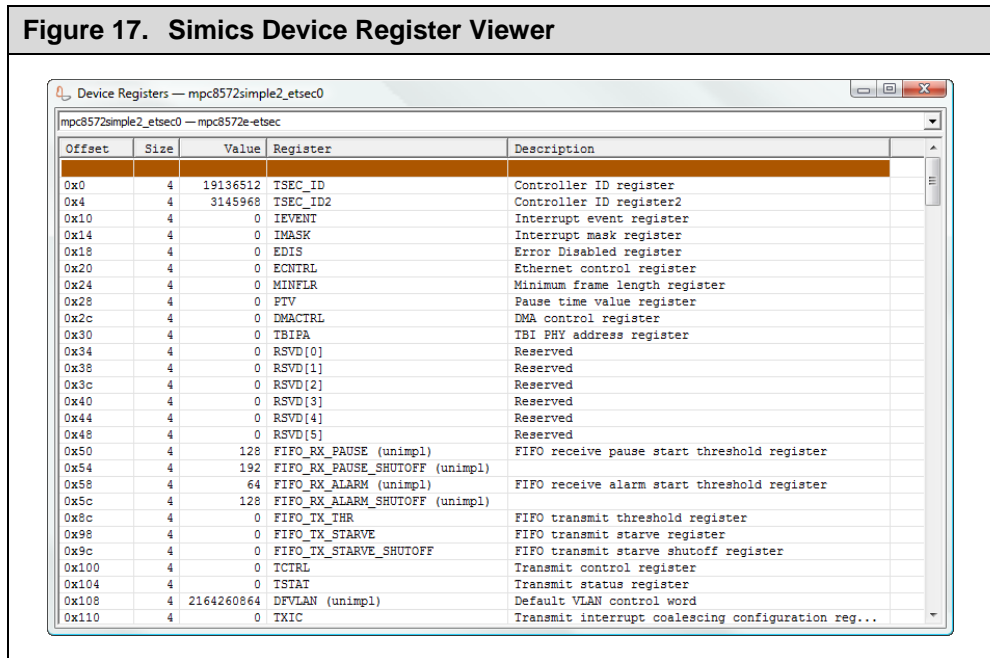
DML code can call into existing C and C++ code. This is useful in order to take advantage of existing libraries for complicated computations, such as compression, crypto, and media coding libraries. It can also be used to wrap functional models for devices already written in C or C++ into DML, and to put a register map onto a basic algorithm model for a hardware accelerator or signal processing block. This is often used to import device kernels written in C and C++ into Simics simulations from the hardware design flow.

There is no need for the existing code to be available in source form, as long as there is a header file and a linkable library file. It is also possible to call into a DML model from C and C++ code, as DML methods correspond to C functions. For example, DML can be used to create a memory-map front-end interface for a function-call driven model of a piece of hardware. It also makes it possible to reuse pieces of functionality from existing TLM models written without framework support or for different simulation APIs.

TOOL SUPPORT

Since DML makes a lot of information about a device explicit and easily accessible to software tools, it provides a good basis for intelligent tool support.

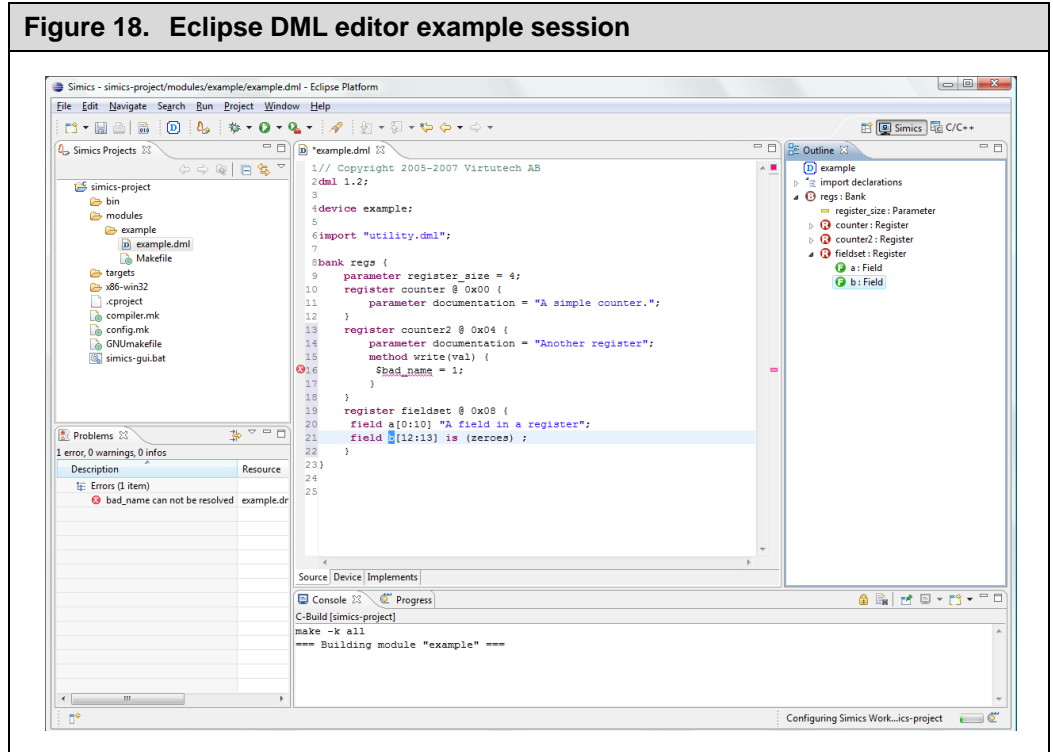
Figure 17. Simics Device Register Viewer



Offset	Size	Value	Register	Description
0x0	4	19136512	TSEC_ID	Controller ID register
0x4	4	3145968	TSEC_ID2	Controller ID register2
0x10	4	0	IEVENT	Interrupt event register
0x14	4	0	IMASK	Interrupt mask register
0x18	4	0	EDIS	Error Disabled register
0x20	4	0	ECNTRL	Ethernet control register
0x24	4	0	MINFLR	Minimum frame length register
0x28	4	0	PTV	Pause time value register
0x2c	4	0	DMACTRL	DMA control register
0x30	4	0	TBIPA	TBI PHY address register
0x34	4	0	RSVD[0]	Reserved
0x38	4	0	RSVD[1]	Reserved
0x3c	4	0	RSVD[2]	Reserved
0x40	4	0	RSVD[3]	Reserved
0x44	4	0	RSVD[4]	Reserved
0x48	4	0	RSVD[5]	Reserved
0x50	4	128	FIFO_RX_PAUSE (unimpl)	FIFO receive pause start threshold register
0x54	4	192	FIFO_RX_PAUSE_SHUTOFF (unimpl)	FIFO receive alarm start threshold register
0x58	4	64	FIFO_RX_ALARM (unimpl)	FIFO receive alarm start threshold register
0x5c	4	128	FIFO_RX_ALARM_SHUTOFF (unimpl)	FIFO receive alarm start threshold register
0x8c	4	0	FIFO_TX_THR	FIFO transmit threshold register
0x98	4	0	FIFO_TX_STARVE	FIFO transmit starve register
0x9c	4	0	FIFO_TX_STARVE_SHUTOFF	FIFO transmit starve shutoff register
0x100	4	0	TCTRL	Transmit control register
0x104	4	0	TSTAT	Transmit status register
0x108	4	2164260864	DFVLAN (unimpl)	Default VLAN control word
0x110	4	0	TXIC	Transmit interrupt coalescing configuration reg...

As Figure 17 shows, Simics extracts the register documentation from DML models and uses it when displaying the register banks of a device with register sizes, offsets, values, names, and inline descriptions. Note that unimplemented and reserved registers are marked. This makes it much easier to quickly check the configuration of a device.

In Simics, DML devices also map programming registers directly to Simics attributes. This makes it easy and to access and change the register contents from the Simics user interface, without having to start memory transactions inside the simulated machine. It also means that Simics checkpoint files are easy to read and edit for a human, since the information is presented in an easy structure.



To support DML programming, Virtutech provides Eclipse and Emacs editor modes customized for DML. Figure 18 shows an example session for the Eclipse DML editor, including such features as device outlines and online grammar checking of the DML code.

Note that the primary goal of DML is to help with device model programming. As a secondary benefit, DML provides a basis for good user interaction with the virtual system. DML was not designed as a vehicle to communicate device design information between EDA tools, even though it could definitely be used to do that.

To help in debugging DML code, the DML compiler inserts line directives in the generated C code, which makes it possible to step DML code using the standard gdb debugger.

SUMMARY

DML is a domain-specific language designed to speed up and simplify the coding of transaction-level device models for fast functional simulators. It achieves this by reducing redundancy in the code and providing purpose-built syntax for common tasks like setting up a register map and connecting devices to each other. It reduces the code size by automatically generating much of the interface code between device models and the simulation infrastructure. It creates fast models by making it harder to make performance mistakes in coding and enforcing a consistent level of abstraction across devices.

CONTACT INFORMATION

North and South America sales_americas@virtutech.com	Europe, Middle-East, Africa sales_emea@virtutech.com
Asia-Pacific sales_apac@virtutech.com	Japan sales_japan@virtutech.com
http://www.virtutech.com	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

TRADEMARKS. Virtutech, Simics, Hindsight, and the logos thereof, are trademarks or registered trademarks of Virtutech, Inc. and/or its subsidiaries, in the United States and/or other countries.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

THIS PUBLICATION COULD INCLUDE TECHNICAL INACCURACIES OR TYPOGRAPHICAL ERRORS. CHANGES ARE PERIODICALLY ADDED TO THE INFORMATION HEREIN; THESE CHANGES WILL BE INCORPORATED IN NEW EDITIONS OF THE PUBLICATION. VIRTUTECH MAY MAKE IMPROVEMENTS AND/OR CHANGES IN THE PRODUCT(S) AND/OR THE PROGRAM(S) DESCRIBED IN THIS PUBLICATION AT ANY TIME.

Virtutech, Inc., 2001 Gateway Place, Suite 201E, San Jose, CA 95110, USA