DML 1.0 Reference Manual

Simics Version	3.0
Revision	1406
Date	2008-02-19

Contents

Chapter 1

Introduction

DML (*Device Modeling Language*) is a programming language for modeling devices in Simics. DML has been designed to make it easy to represent the kind of things that are needed by a device model, and uses special syntactic constructs to describe common elements such as memory mapped hardware registers and connections to other Simics configuration objects.

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.

This document describes the DML language, the standard libraries, and the `dmlc` compiler, as of version 1.0 of DML. See also the *DML Tutorial* for a more hands-on introduction to DML, and the *Device Modeling Guidelines* document for a broader 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.

Chapter 2

Running dmlc

The syntax for running `dmlc` from the command line is:

```
dmlc [options] input [output-base]
```

where *input* should be the name of a DML source file. If *output-base* is provided, it will be used to name the created files. The name of a DML source file should in general have the suffix "`.dml`". The *input* file must contain a `device` declaration; see Section 4.4.2.

The output of `dmlc` is a set of C source and header files that can be compiled into a Simics module in the same way a hand-written module would. (See the *Simics Programming Guide* for details on writing Simics modules in C.) The generated C code should be possible to compile with any ISO C compatible compiler. For reference and debugging purposes, an HTML file summarizing the modeled device is also generated.

2.1 Command-line Options

The following are the available command-line options to `dmlc`:

-I *path*
Add *path* to the search path for imported modules.

-D *name=definition*
Define a compile-time parameter.

-M
Output makefile rules describing dependencies.

-m
Build a Simics module. This makes dmlc include the necessary glue code in the generated C file to be able to compile it as a Simics module.

-T
Show tags on warning messages. The tags can be used with the `--nowarn` and `--warn` options.

--help
Print a short help text and exit.

--nowarn=*tag*
Suppress selected warnings. The tags can be found using the `-T` option.

--warn=*tag*
Enable selected warnings. The tags can be found using the `-T` option.

--noline
Suppress line directives for the C preprocessor so that the C code can be debugged.

--make-syntax={native, cygwin}
Windows only. Specify path syntax to be used when generating makefile rules (using the -M) flag. The default is "cygwin", which generates makefile rules are to be used with Cygwin's GNU make. If the syntax is set to "native", paths will be converted to native form, suitable for use with MinGW's make.

The flag is ignored on non-Windows hosts.

--html
Enable the output of an HTML file describing register layout.

--dump
Enable the output of an dump file describing the structure of the device model.

--version
Print the version of dmlc and exit.

Chapter 3

The Object Model

DML is structured around an *object model*, where each DML program describes a single *device class* (being a subclass of the built-in generic device class), which can contain a number of *member objects*. Each member object can in its turn have a number of members of its own, and so on in a nested fashion.

Every object (including the device itself) can also have *methods*, which implement the functionality of the object, and *parameters*, which are constant-valued members that exist only during compilation.

Each object is an instance of a *class*, often simply called the "object type"; the built-in object types are described in Section 3.2. There is no way of adding user-defined classes in DML 1.0. However, each object is in general locally modified by adding (or overriding) members, methods and parameters - this can be viewed as creating a local one-shot subclass for each instance.

Many parts of the DML object model are automatically mapped onto the Simics "configuration object" model; most importantly, the device class itself, its attributes and interface methods. (See the *Simics User Guide* and the *Simics Programming Guide* for details.)

3.1 Device Structure

A device is made up of a number of member objects and methods, where any object may contain further objects and methods of its own. Although some object types, such as **events**, are generic and can appear almost anywhere, many object types only make sense in a particular context and are not allowed elsewhere:

- Objects of type **attribute**, **bank**, **implement**, or **connect** (or **array of connect**), may only appear as components of a **device**.
- Objects of type **register** (or **array of register**) may only appear as components of a **bank**, or indirectly as a component of a **group** or **array of group** that is (directly or indirectly) a component of a **bank**.
- Objects of type **field** may only appear as components of a **register**.
- Objects of type **interface** may only appear as components of a **connect**.

3.2 Object Types

The following is an overview of the available *object types* (the built-in classes):

device
: Each DML program defines a subclass of the **device** class, which can be instantiated as a configuration object in Simics.

attribute
: An arbitrary Simics configuration-object attribute of the **device**. Many attributes are created automatically from the declaration of device members such as registers and connects, but it is possible to manually add other attributes as well. This can be used to provide synthetic information about the device for inspection, or as a simple programming interface. Attributes are typically saved automatically when a Simics checkpoint is created.

bank
: A register bank. Banks make sets of registers accessible by placing them in an *address space*. Register banks can be individually mapped into Simics memory spaces by using a separate *function number* for each bank. The **bank** class is a subclass of **group**.

register
: A **register** holds an integer value, and is generally used to model a hardware register. The width of a register is a nonzero number of bytes (a register may not be wider than 8 bytes). Registers may only occur within a **bank** (directly or indirectly). Registers divide the address space of a bank into discrete elements with non-overlapping addresses. Registers are used for communication via memory-mapped I/O, and also for storing device state in natural way. Sometimes it is useful to have registers that belong to a bank but are not mapped into its address space.

array of register
: Register arrays are used for compact representation of related registers within the same bank.

field
: A **register** can be further divided into *fields* on the bit level. Each such field can be accessed separately, both for reading and for writing. The fields of a register may not overlap.

group
: Groups are generic container objects, and are mainly used to define logical groups of registers. See also **bank**.

array of group
: Arrays of groups are useful for modeling repeated groups of registers.

connect
: A **connect** object holds a reference to a Simics configuration object. (Typically, the connected object is expected to implement some particular Simics-interface.) An attribute

with the same name is added to the **device**; thus, a **connect** is similar to a simple **attribute** object. Usually, initialization is done from outside the device, e.g., in a Simics configuration file.

array of connect
Connect-arrays make it easy to connect a whole array of objects.

interface
An **interface** object may only occur within a **connect**. It is used to declare a Simics-interface assumed to be implemented by the connected object. In many cases, the name of the interface is sufficient, and the body of the object can be left empty.

implement
An **implement** object is used to define a Simics-interface that the **device** implements. The methods defined within the **implement** must correspond to the functions of the interface.

event
An **event** object is an encapsulation of a Simics event that can be posted on a time or step queue. Almost all objects can have event members, except events themselves, and objects of type **field** and **implement**.

3.3 Parameters

Parameters are *compile-time, constant-valued* object members. They exist only during translation from DML to C; no storage is allocated for parameters, and they cannot be updated - only overridden.

No type declarations are necessary for parameters; all DML compile-time computations are dynamically type-checked. The possible types of values of a parameter are listed below.

Parameters are used to describe static properties of the objects, such as names, sizes, and offsets. There are a number of standard parameters that are always defined for every object, and for each object type there is a set of additional pre-defined parameters. Furthermore, the programmer can add any number of new parameters, or override the pre-defined ones.

3.3.1 Parameter Types

A parameter can be assigned a value of one of the following types:

integer
An arbitrarily-sized integer value.

float
A floating-point value.

string
A sequence of characters. String literals are written within double quotes. Strings can be split over several lines using the + operator; see Section 4.12.10.

bool

One of the values `true` and `false`. These can only be used in tests and boolean expressions; they are not integer values.

list

A list of arbitrary values. Lists are written as `[x1, ..., xN]`.

reference

A reference to a DML object or method.

undefined

The value `undefined` is a unique value which can only be used in the compile-time test `defined` *x*. The result of the test is `false` if *x* has the value `undefined`, and `true` otherwise. The `undefined` value is used mainly as a default for parameters that are intended to be overridden.

3.4 Methods

Methods are object members providing implementation of the functionality of the object. Although similar to C functions, DML methods can have both a number of input parameters and zero, one or more output parameters. DML methods also support a basic exception handling mechanism using `throw` and `try`.

3.5 Devices

A *device* defined by a DML program corresponds directly to a Simics *configuration object*, i.e., a component that can be included in a Simics configuration. Typically, the device object will make its functionality available by implementing one or more Simics API *interfaces* (sets of methods).

A configuration object can also contain *state* (data), and can have any number of *attributes*, which are a specialized form of interfaces with a pair of ***get*/*set*** methods used for inspecting and manipulating some part of the state. (See the *Simics User Guide* and *Simics Programming Guide* for further details about configuration objects, interfaces and attributes.)

3.5.1 Memory-mapped I/O

The most common interface that a device might implement is the `io_memory` interface, which allows the device to be mapped into a Simics memory space.

To use the standard implementation of the `io_memory` interface in a DML file, include the following line:

```
import "io-memory.dml";
```

The standard library file `io-memory.dml` provides the default implementation of the `io_memory` interface, in such a way that each register bank of the device for which the *function* parameter is set (to a nonnegative integer), can be easily mapped into a memory space in a Simics configuration by specifying the device object and the function number. Accesses to the mapped addresses are automatically translated into calls to the ***access*** method of the corresponding bank.

3.5.2 Connections to other objects

Interaction with other Simics configuration objects from within a device model is made easy through **connect** objects in DML, which literally serve as connectors, and can be initialized from outside the device - typically in a configuration file. Connected objects are generally expected to implement some particular interface so that the device can communicate with them.

A device can also implement Simics interfaces on its own, in addition to the standard `io_memory` interface described above. This can be done by defining an **implement** object containing the necessary methods. The device will be automatically be registered in Simics as implementing the interface.

3.5.3 Attributes

The Simics configuration object attributes that are automatically added to the device object for components such as connects and registers are often sufficient, but is also possible to define arbitrary additional attributes of the device, represented by **attribute** objects.

3.6 Register Banks

A *register bank* (or simply *bank*) is an abstraction that is is used to group *registers* in DML, and is the unit for memory mapped communication via the *io_memory* interface (see Section 3.5.1).

For a register bank to be mapped into a Simics memory space, its `function` parameter must be set to a nonnegative integer that is unique within the device. The function number is used when setting up a memory space mapping in Simics, and allows a single device object to be configured with any number of address ranges mapped onto different banks (possibly with more than one address range mapped to each single bank).

A bank normally contains a number of **register** objects, but that is not a requirement. It is possible to define an alternative implementation of the ***access*** method of a particular bank, that does not depend on registers.

3.7 Registers

A *register* is an component of a register bank that contains an integer value, which can be either unsigned (the default) or signed. Normally, a register corresponds to a segment of consecutive locations in the address space of the bank; however, it is also possible (and often useful) to have registers that are not mapped to any address within the bank.

Every register has a fixed *size* (or *width*), which is an integral, nonzero number of 8-bit bytes. A single register cannot be wider than 8 bytes, in DML 1.0. The size of the register is given by the `size` parameter, which can be specified either by a normal parameter assignment, as in

```
register r1 {
  parameter size = 4;
  ...
}
```

or more commonly, using the following short-hand syntax:

```
register r1 size 4 {
  ...
}
```

which has the same meaning. The default size is provided by the `register_size` parameter of the containing register bank, if that is defined.

The value of the register can be accessed directly by referencing the object, as in

```
log "info": "the value of register r1 is %d", $r1;
```

or

```
$r1 += 1;
```

Storage for the register value is created by default in the generated C code. If this is not needed (e.g., if write accesses to the register are ignored and read accesses always return a constant value such as 0), it can be disabled in order to save memory by setting the `allocate` parameter to `false`. Non-allocated registers cannot be accessed by value.

3.7.1 Mapping Addresses To Registers

For a register to be mapped into the internal address space of the containing bank, its starting address within the bank must be given by setting the `offset` parameter. The address range occupied by the register is then from `offset` to `offset` + `size` - 1. The offset can be specified by a normal parameter assignment, as in

```
register r1 {
  parameter offset = 0x0100;
  ...
}
```

or using the following short-hand syntax:

```
register r1 @ 0x0100 {
  ...
}
```

similar to the `size` parameter above. Usually, a normal read/write register does not need any additional specifications apart from the size and offset, and can simply be written like this:

```
register r1 size 4 @ 0x0100;
```

or, if the bank contains several registers of the same size:

```
bank b1 {
  parameter register_size = 4;
  register r1 @ 0x0100;
  register r2 @ 0x0103;
  ...
}
```

The translation from the bank address space to the actual value of the register is controlled by the `byte_order` parameter. When it is set to `"little-endian"` (the default), the lowest address, i.e., that defined by `offset`, corresponds to the least significant byte in the register, and when set to `"big-endian"`, the lowest address corresponds to the most significant byte in the register. The third allowed value of this parameter is `undefined`, which ensures that neither assumption is ever made.

3.7.2 Register Attributes

For every register, an attribute of integer type is automatically added to the Simics configuration class generated from the device model. The name of the attribute corresponds to the hierarchy in the DML model; e.g., a register named `r1` in a bank named `bank0` will get a corresponding attribute named *bank0_r1*.

The register value is automatically saved when Simics creates a checkpoint, unless the `configuration` parameter indicates otherwise.

3.7.3 Fields

Real hardware registers often have a number of *fields* with separate meaning. For example, the lowest three bits of the register could be a status code, the next six bits could be a set of flags, and the rest of the bits could be reserved.

To make this easy to express, a **register** object can contain a number of **field** objects. Each field is defined to correspond to a bit range of the containing register. (Technically, a register that does not contain any explicitly defined fields automatically gets a single, anonymous field, which covers the whole register.)

Only the values of the fields are stored in the generated C code; the value of the register as a whole is composed from the field values when needed (for example, when creating a Simics checkpoint). Storage for individual fields can be controlled by the `allocate` parameter, which by default is inherited from the **register** object.

For example, the register described above could be modeled as follows, using the default little-endian bit numbering.

```
bank b2 {
  register r0 size 4 @ 0x0000 {
    field status   [2:0];
    field flags    [8:3];
    field reserved [15:9] {
      parameter allocate = false;
    } ;
  }
  ...
}
```

Note that the most significant bit number is always the first number (to the left of the colon) in the range, regardless of whether little-endian or big-endian bit numbering is used. (The bit numbering convention used in a source file can be selected by a `bitorder` declaration.)

The value of the field can be accessed by referencing the object, as for a register, e.g.:

```
log "info": "the value of the status field is %d", $r0.status;
```

Note however that non-allocated fields cannot be accessed this way, since they do not have a stored value.

3.8 Attributes

An *attribute* object in DML represents a Simics configuration object attribute of the device. An attribute is basically a name with an associated pair of ***get*** and ***set*** functions; see Section 5.2.3. The type of the value read and written through the get/set functions is controlled by the `type` parameter; see Section 5.1.4. More information about configuration object attributes can be found in the *Simics Programming Guide*.

Attributes can be used to provide synthetic information about the device for inspection, or as a simple programming interface. By default, attributes are saved automatically when a Simics checkpoint is created; this is controlled by the the `configuration` parameter.

Note that an attribute is not required to save the written value; this is entirely dependent on the intended semantics of the attribute. For example, a very simple attribute could do nothing in the ***set*** function, and always return zero from the ***get*** function.

By defining the `allocate_type` parameter, the ***get*** and ***set*** functions and the `type` parameter can be created automatically, for simple attribute types that store the written value.

For attributes defined using `allocate_type`, the value of the attribute can be accessed within DML by referencing the object directly, e.g.:

```
log "info": "the value of attribute a is %d", $dev.a;
```

3.9 Connects

A *connect* object is a container for a reference to an arbitrary Simics configuration object. An attribute with the same name as the connect, that can be assigned a value of type "Simics object", is added to the Simics configuration class generated from the device model. A connect is thus similar to a simple **attribute** object.

Typically, the connected object is expected to implement one or more particular Simics-interfaces, such as `simple_interrupt` (see the *Simics Reference Manual* for details).

Initialization of the connect (i.e., setting the object reference) is typically done from outside the device, usually in a Simics configuration file. As for attributes, the parameter `configuration` controls whether the value must be initialized when the object is created, and whether it is automatically saved when a checkpoint is created; see Section 5.1.8.

3.9.1 Interfaces

In order to use the Simics interfaces that a connected object implements, they must be declared within the **connect**. This is done through *interface* objects, which name the expected interfaces and may also specify additional properties.

An important property of an interface object is whether or not a connected object is *required* to implement the interface. This can be controlled through the interface parameter `required`, which is `true` by default; see Section 5.1.9. Attempting to connect an object that does not fulfill the requirement will cause a runtime error.

By default, the C type of the Simics interface corresponding to a particular interface object is assumed to be the name of the object itself with the string `"_interface_t"` appended. (The C type is typically a `typedef`:ed name for a struct containing function pointers). This can be changed by specifying an explicit `c_type` parameter in the interface object.

The following is an example of a connect with two interfaces, one of which is not required:

```
connect plugin {
  interface serial_device;
  interface rs232_device { parameter required = false; }
}
```

3.10 Implements

When a device needs to export Simics interfaces (such as the `io_memory` interface defined in the standard library file `io-memory.dml`), this is specified by an **implement** object, containing the methods that implement the interface. The name of the object is also used as the name of the Simics interface registered for the generated device, and the names and signatures of the methods must correspond to the C functions of the Simics interface. (A device object pointer is automatically added as the first parameter of the generated C functions.)

By default, the C type of the Simics interface is assumed to be the name of the object itself with the string `"_interface_t"` appended. (The C type is typically a `typedef`:ed name for a struct containing function pointers.) This can be changed by specifying an explicit `c_type` parameter in the interface object.

For example, to create an alternative implementation of `io_memory`, we can write:

```
implement io_memory {
  method map(addr_space_t sp, map_info_t info)
         -> (int status)
  {
    ...
  }
  method operation (generic_transaction_t *op, map_info_t info)
         -> (exception_type_t exc)
  {
    ...
  }
}
```

The code generated by `dmlc` for this example assumes that the `io_memory_interface_t` is a C struct type with fields `map` and `operation` that can hold function pointers of the corresponding types. (The `io_memory_interface_t` type is defined in the standard library file `simics-api.dml`, which is automatically included by the `dmlc` compiler.)

3.11 Events

An *event* object is an encapsulation of a Simics event that can be posted on a processor time or step queue. The location of event objects in the object hierarchy of the device is not important, so an event object can generally be placed wherever it is most convenient.

An event has a built-in ***post*** method, which inserts the event in the default queue associated with the device. When the event is triggered, the ***event*** method of the event is called. The built-in default ***event*** method simply logs that the event has occurred, which can be useful for testing purposes.

Chapter 4

Device Modeling Language 1.0

This chapter describes the DML language, version 1.0. It will help to have read and understood the object model in the previous chapter before reading this chapter.

4.1 Overview

DML is not a general-purpose programming language, but a modeling language, targeted at writing Simics device models. The algorithmic part of the language is an extended subset of ISO C; however, the main power of DML is derived from its simple object-oriented constructs for defining and accessing the static data structures that a device model requires, and the automatic creation of bindings to Simics.

Furthermore, DML provides syntax for *bit-slicing*, which much simplifies the manipulation of bit fields in integers; `new` and `delete` operators for allocating and deallocating memory; a basic `try`/`throw` mechanism for error handling; built-in `log` and `assert` statements; and a powerful metaprogramming facility using *templates*, method inlining, and compile-time evaluation for shaping the generated code.

Most of the built-in Simics-specific logic is implemented directly in DML, in standard library modules that are automatically imported; the `dmlc` compiler itself contains as little knowledge as possible about the specifics of Simics.

4.2 Lexical Structure

For the most part, the lexical structure of DML resembles that of C. However, DML distinguishes between "object context" and "C context", so that some C keywords such as `register`, `signed` and `unsigned`, are allowed as identifiers in object context, while on the other hand many words that look like keywords in the object context, such as `bank`, `event` and `data`, are in fact allowed as identifiers in all contexts.

Another major difference from C is that names do not generally need to be defined before their first use. This is quite useful, but might sometimes appear confusing to C programmers.

Reserved words

All ISO/ANSI C reserved words are reserved words in DML (even if currently unused). In addition, the C99 and C++ reserved words `restrict`, `inline`, `this`, `new`, `delete`, `throw`, `try`, `catch`, and `template` are also reserved in DML. The C++ reserved words `class`, `namespace`, `private`, `protected`, `public`, `using`, and `virtual`, are reserved in DML for future use.

The following words are reserved specially by DML: `after`, `assert`, `call`, `cast`, `defined`, `error`, `foreach`, `in`, `is`, `local`, `log`, `parameter`, `select`, `sizeoftype`, `typeof`, `undefined`, `vect`, and `where`.

Identifiers
Identifiers in DML are defined as in C; an identifier may begin with a letter or underscore, followed by any number of letters, numbers, or underscores. Within C context, object-context identifiers are always prefixed by a `$` character.

Constant Literals
DML does not support octal constants (written with a leading `0` in C); however, DML adds binary constants, written "`0b...`", as in `0b1111` or `0b11000011`.

Comments
Both ordinary C-style (`/*...*/`) comments and C++/C99 (`//...`) comments are allowed in DML.

4.3 Module System

DML employs a very simple module system, where a *module* is any source file that can be imported using the `import` directive. Such files may not contain a `device` declaration, but otherwise look like normal DML source files. The imported modules are merged into the main program as if all the code was contained in a single file (with some exceptions). This is similar to C preprocessor `#include` directives, but in DML each imported file must be possible to parse in isolation, and may contain declarations (such as `bitorder`) that are only effective for that file.

4.4 Source File Structure

A DML source file describes both the structure of the modeled device and the actions to be taken when the device is accessed.

A DML source file defining a device starts with a *language version declaration* and a *device declaration*. After that, any number of *parameter declarations*, *methods*, *data fields*, *object declarations*, or *global declarations* can be written. A DML file intended to be *imported* (by an `import` statement in another DML file) has the same layout except for the device declaration.

4.4.1 Language Version Declaration

Every DML source file should contain a version declaration, on the form "`dml` *m*.*n*;", where *m* and *n* are nonnegative integers denoting the major and minor version of DML, respectively, that the source file is written in. The version declaration allows the `dmlc` compiler to select the proper versions of the DML parser and standard libraries to be used for the file. (Note that each file has its own individual language version, even if it is imported by a file using another version.) A file should not import a file with a higher language version than its own.

The version declaration must be the first declaration in the file, possibly preceded by comments. For example:

```
// My Device
dml 1.0;
...
```

A file that does not contain any version declaration is assumed to have the version DML 0.9, for reasons of backward compatibility with existing legacy code, but this feature will eventually be deprecated. All new code should use DML 1.0 or later, and all old code should at a minimum be given the version declaration “dml 0.9;”, or preferably be rewritten in DML 1.0.

4.4.2 Device Declaration

Every DML source file that contains a device declaration is a *DML program*, and defines a device class with the specified name. Such a file may *import* other files, but only the initial file may contain a device declaration.

The device declaration must be the first proper declaration in the file, only preceded by comments and the language version declaration. For example:

```
/*
 *  My New Device
 */
dml 1.0;
device my_device;
...
```

4.4.3 Source File Example

The following is an example of a small DML program defining a very simple device. This lacks many details that would appear in a real device.

```
dml 1.0;

device excalibur;

connect bus {
  interface pci;
}

bank config_registers {
    parameter function = 0;
    register cfg1 size 4 @ 0x0000 {
        field status {
            method read { ... }
            method write { ... }
        }
        field enable {
          method read { ... }
          method write { ... }
        }
    }
}

bank databank {
    parameter function = 1;
    register r1 size 4 @ 0x0000 {
        field f1 {
            method read { ... }
            method write { ... }
        }
    }
    register r2 size 4 @ 0x0004 {
        field f2 {
            method read { ... }
            method write { ... }
        }
    }
}
```

4.5 Parameter Declarations

A parameter declaration has the general form "`parameter` *name specification*`;`", where *specification* is either "`=` *expr*" or "`default` *expr*". For example:

```
    parameter function = 1;
    parameter byte_order default "little-endian";
```

A default value is overridden by an assignment (=). There can be at most one assignment, and at most one default value, for each parameter. Typically, a default value for a parameter is specified in a template, and the programmer may then choose to override it where the template is used. See Section 4.9.2 for more information about templates.

The *specification* part is in fact optional; if omitted, it means that the parameter is declared to exist (and *must* be given a value, or the program will not compile). This is sometimes useful in templates, as in:

```
template constant {
    parameter value;
    method get -> (v) {
        v = $value;
    }
}
```

so that wherever the template `constant` is used, the programmer is also forced to define the parameter `value`. E.g.:

```
register r0 size 2 @ 0x0000 is (constant) {
    parameter value = 0xffff;
}
```

Note that simply leaving out the parameter declaration from the template definition can have unwanted effects if the programmer forgets to specify its value where the template is used. At best, it will only cause a more obscure error message, such as “unknown identifier”; at worst, the scoping rules will select an unrelated definition of the same parameter name.

You may see the following special form in some standard library files:

```
parameter name auto;
```

for example,

```
parameter parent auto;
```

This is used to explicitly declare the built-in automatic parameters, and should never be used outside the libraries.

4.6 Methods

Methods are similar to C functions, but also have an implicit (invisible) parameter which allows them to refer to the current device instance, i.e., the Simics configuration object representing the device. Methods also support exception handling in DML, using `try` and `throw`. The body of the method is a compound statement in an extended subset of C; see also Section 4.10. It is an error to have more than one method declaration using the same name within the same scope.

4.6.1 Input and Output Parameters

In addition to the input parameters, a DML method can have any number of output parameters, in contrast to C functions which have at most one return value. DML methods do not use the keyword `void` - an empty pair of parentheses always means “zero parameters”, and can even be omitted. Apart from this, the parameter declarations of a method are ordinary C-style declarations.

For example,

-
  ```
  method m1 {...}
  ```

  and

  ```
  method m1() {...}
  ```

  are equivalent, and define a method that takes no input parameters and returns nothing.

-
  ```
  method m2(int a) -> () {...}
  ```

  defines a method that takes a single input parameter, and also returns nothing.

-
  ```
  method m3(int a, int b) -> (int c) {
    ...;
    c = ...;
  }
  ```

  defines a method with two input parameters and a single output parameter. Output parameters must be explicitly assigned before the method returns.

-
  ```
  method m4() -> (int c, int d) {
    ...;
    c = ...;
    d = ...;
    return;
  }
  ```

  has no input parameters, but two output parameters. Ending the method body with a `return` statement is optional.

4.6.2 Calling Methods

In DML, a method call is performed with one of the `call` and `inline` keywords. (In DML 1.0, method calls can only be statements, not expressions; this limitation may be removed in later versions of DML.) For instance,

```
call $access(...) -> (a, b)
```

will call the method 'access' in the same object, assigning the values of the output parameters to variables a and b. (Note the '$' character which is necessary for referring to the method.) The call might be inline expanded depending on the C compiler used to compile the generated code, but there is no guarantee.

On the other hand,

```
inline $access(...) -> (a, b)
```

has the same semantics as the `call`, but will always inline expand the called method.

4.6.3 Methods as Macros

Methods can also be used as macros, by omitting the types on one or more of the input parameters. A method defined this way can only be called through an `inline` statement; see Section 4.11.5.

For example,

```
method twice(x) -> (y) { y = x + x; }
```

could be used to double any numeric value without forcing the result to be of any particular type, as in

```
int32_t x;
int64_t y;
inline twice(x) -> (x);
inline twice(y) -> (y);
```

This is sometimes referred to as a *polymorphic* method.

This form of macros is particularly useful when writing *templates* (see Section 4.9.2). Note that the name scoping rules and the semantics of calls are the same as for normal methods; in other words, it is a form of "hygienic macros".

4.6.4 External Methods

A method can be declared *external*, which means that a C function corresponding to the method is guaranteed to be generated, and will have external linkage. This is done by prefixing the name of the method with the keyword `extern`; e.g.:

```
method extern my_method(int x) { ... }
```

An external method must have a proper signature, i.e., the types of all its input and output parameters must be specified.

External methods are rarely used, since most of the needs for making DML methods accessible from outside the device model itself are covered by the **implement**, **event**, and **attribute** type objects.

4.7 Data Fields

A *data* declaration creates a named storage location for an arbitrary run-time value. The name belongs to the same namespace as objects and methods. The general form is:

```
data declaration;
```

where *declaration* is similar to a C variable declaration; for example,

```
data int id;
data bool active;
data double table[16];
data conf_object_t *obj;
```

4.8 Object Declarations

The general form of an object declaration is "*type name extras desc* `{ ... }`" or "*type name extras desc*`;`", where *type* is an object type such as `bank`, *name* is an identifier naming the object, and *extras* is optional special notation which depends on the object type. *desc* is an optional string constant giving a very short summary of the object. Ending the declaration with a semicolon is equivalent to ending with an empty pair of braces. The *body* (the section within the braces) may contain *parameter declarations*, *methods*, *data fields*, and *object declarations*.

For example, a `register` object may be declared as

```
register r0 @ 0x0100 "general-purpose register 0";
```

where the "`@` *offset*" notation is particular for the **register** object type; see below for details.

An object declaration with a *desc* section, on the form

```
type name ... desc { ... }
```

is equivalent to defining the parameter `desc`, as in

```
type name ... {
    parameter desc = desc;
    ...
}
```

In the following sections, we will leave out *desc* from the object declarations, since it is always optional. Another parameter, `documentation` (for which there is no short-hand), may also be defined for each object, and is used to give a more detailed description. See Section 5.1.1 for details.)

If two object declarations with the same name occur within the same containing object, and they specify the same object type, then the declarations are concatenated; e.g.,

```
bank b {
    register r size 4 { ...body1... }
    ...
    register r @ 0x0100 { ...body2... }
    ...
}
```

is equivalent to

```
bank b {
    register r size 4 @ 0x0100  {
        ...body1...
        ...body2...
    }
    ...
}
```

However, it is an error if the object types should differ.

Some object types (**register**, **group**, **attribute**, and **connect**) may be used in *arrays*. The general form of an object array declaration is

```
type name [size] extras { ... }
```

where *size* is the number of elements in the array. The size must be a compile time constant. For instance,

```
register regs[16] size 2 {
    parameter offset = 0x0100 + 2 * $i;
    ...
}
```

or written more compactly

```
register regs[16] size 2 @ 0x0100 + 2 * $i;
```

defines an array named `regs` of 16 registers (numbered from 0 to 15) of 2 bytes each, whose offsets start at 0x0100. There is also a special syntax "*type name* `[`*variable* `in 0..`*max*`] ...`", where *max* = *size* `- 1`. See Section 5.1.2 for details about arrays and index parameters.

The following sections give further details on declarations for object types that have special conventions.

4.8.1 Bank Declarations

The general form of a **bank** declaration is

```
bank name { ... }
```

where *name* may be omitted. The elements (e.g., registers) of a bank that has no name belong to the namespace of the parent object, i.e., the device. There is at most one such anonymous bank object per device; multiple "`bank { ... }`" declarations are concatenated.

4.8.2 Register Declarations

The general form of a **register** declaration is

```
register name size n @ d is (templates) { ... }
```

Each of the "`size` *n*", "`@` *d*", and "`is (`*templates*`)`" sections is optional, but if present, they must be specified in the above order.

- A declaration

  ```
  register name size n ... { ... }
  ```

  is equivalent to

  ```
  register name ... { parameter size = n; ... }
  ```

- A declaration

  ```
  register name ... @ d ... { ... }
  ```

  is equivalent to

  ```
  register name  ... { parameter offset = d; ... }
  ```

- A declaration

  ```
  register name ... is (t1,...,tN) { ... }
  ```

is equivalent to

```
register name ... { is t1; ... is tN; ... }
```

Templates are further described in Section 4.9.2.

4.8.3 Field Declarations

The general form of a **field** declaration is

```
field name [highbit:lowbit] is (templates) { ... }
```

or simply

```
field name [bit] ... { ... }
```

specifying a range of bits of the containing register, where the syntax `[`*bit*`]` is short for `[`*bit*`:`*bit*`]`. Both the "`[...]`" and the `is (`*templates*`)` sections are optional; in fact, the "`[...]`" syntax is merely a much more convenient way of defining the (required) field parameters `lsb` and `msb` (cf. Section 5.1.7).

- A declaration

  ```
  field name [high:low] is (t1,...,tN) { ... }
  ```

  is equivalent to

  ```
  field name [high:low] { is t1; ... is tN; ... }
  ```

For a range of two or more bits, the first (leftmost) number always indicates the *most significant bit*, regardless of the bit numbering scheme used in the file. This corresponds to how bit fields are usually visualized, with the most significant bit to the left.

The bits of a register are always numbered from zero to *n* - 1, where *n* is the width of the register. If the default little-endian bit numbering is used, the least significant bit has index zero, and the most significant bit has index *n* - 1. In this case, a 32-bit register with two fields corresponding to the high and low half-words may be declared as

```
register HL size 4 ... {
    field H [31:16];
    field L [15:0];
}
```

If instead big-endian bit numbering is selected in the file, e.g., with a "`bitorder be32;`" declaration, the most significant bit has index zero, and the least significant bit has the highest index. In that case, the register above may be written as

```
register HL size 4 ... {
    field H [0:15];
    field L [16:31];
}
```

This is useful when modeling a system where the documentation uses big-endian bit numbering, so it can be compared directly to the model.

Note that for a bit field declaration, only the "endianism" selected by a `bitorder` declaration in the file has any effect; the base width is always that of the containing register, so in a 16-bit register, big-endian bit 0 corresponds to little-endian bit 15, but in a 32-bit register, big-endian bit 0 corresponds to little-endian bit 31.

4.9 Global Declarations

The following sections describe the global declarations in DML. These can only occur on the top level of a DML program, i.e., not within an object or method. Unless otherwise noted, their scope is the entire program.

4.9.1 Import Declarations

`import` *filename*`;`

Imports the contents of the named file. *filename* must be a string literal, such as `"utility.dml"`. The `-I` option to the `dmlc` compiler can be used to specify directories to be searched for import files.

Note that imported files are parsed as separate units, and use their own language version and bit order declarations.

4.9.2 Template Declarations

`template` *name desc* `{ ... }`

Defines a named *template* - a piece of code that can be reused in multiple locations. This is similar to a C preprocessor macro; however, unlike macros, the body cannot consist of arbitrary text, but must have the same form as an object declaration body. Also, DML templates do not take any arguments; instead, the generic DML parameter system is used for such purposes.

Templates are instantiated using `is` declarations, written as

```
    is name;
```

These can be used in any context where an object declaration may be written, and has the effect of expanding the body of the template at the point of the `is`. Note that the expansion is purely textual, so e.g., two templates which define methods with the same name cannot both be used in the same context.

As in an object declaration, unless the *desc* string is omitted, it is equivalent to defining the parameter `desc`, as in

```
    template name {
        parameter desc = desc;
        ...
    }
```

Hence, the description does not pertain to the template in itself, but to any object which uses the template.

4.9.3 Bitorder Declarations

```
bitorder name;
```

Selects the default bit order to be used for interpreting bit-slicing expressions and bit field declarations in the file. *name* is one of the identifiers `le`, `be16`, `be32`, and `be64`, implying either little-endian order with arbitrary default width (`le`), or big-endian order with a fixed default width in bits (`be...`). Only the listed widths are allowed.

A `bitorder` declaration, if present, must follow immediately after the `device` declaration, preceding any other global declarations. The scope of the declaration is the whole of the file it occurs in. If no `bitorder` declaration is present in a file, the default bit order is `le` (little-endian). The bitorder does not extend to imported files; for example, if a file containing a declaration "`bitorder be32;`" imports a file with no bit order declaration, the latter file will still use the default `le` order.

Note that it is difficult to write general bit-manipulating methods for arbitrary-width data, if big-endian conventions are used. It is often a better idea to place such code in a separate file which uses little-endian order, and then import that file from a big-endian context.

4.9.4 Constant Declarations

```
constant name = expr;
```

Defines a named constant which can be used in C context. *expr* must be a constant-valued expression.

4.9.5 Loggroup Declarations

`loggroup` *name*`;`

Defines a log group, for use in `log` statements. More generally, the identifier *name* is bound to an unsigned integer value that is a power of 2, and can be used anywhere in C context; this is similar to a `constant` declaration, but the value is allocated automatically so that all log groups are represented by distinct powers of 2 and can be combined with bitwise *or* (see Section 4.11.7).

4.9.6 Typedef Declarations

`typedef` *declaration*`;`

Defines a name for a data type, as in C.

4.9.7 Struct Declarations

`struct` *name* `{` *declarations* `}`

Declares a structure datatype with the given name; this is similar to a C `struct` declaration, but in DML, there is no separate namespace for structs. The effect is like that of writing "`typedef struct { ... }` *name*;" in C, although in DML, *name* can also be used for recursive definitions within the struct, as in

```
struct my_struct {
    my_struct *next;
    ...
}
```

4.9.8 Extern Declarations

`extern` *declaration*`;`

Declares an external identifier, similar to a C `extern` declaration; for example,

```
extern int;
extern char *motd;
extern double table[16];
extern conf_object_t *obj;
extern int foo(int x);
```

```
    extern int printf(const char *format, ...);
```

4.9.9 Header Declarations

```
header %{
...
%}
```

Specifies a section of C code which will be included verbatim in the generated C header file for the device. There must be no whitespace between the `%` and the corresponding brace in the `%{` and `%}` markers. The contents of the header section are not examined in any way by the `dmlc` compiler; declarations made in C code must also be specified separately in the DML code proper.

This feature should only be used to solve problems that cannot easily be handled directly in DML. It is most often used to make the generated code include particular C header files, as in:

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

See also `footer` declarations, below.

4.9.10 Footer Declarations

```
footer %{
...
%}
```

Specifies a piece of C code which will be included verbatim at the end of the generated code for the device. There must be no whitespace between the `%` and the corresponding brace in the `%{` and `%}` markers. The contents of the footer section are not examined in any way by the `dmlc` compiler.

This feature should only be used to solve problems that cannot easily be handled directly in DML. See also `header` declarations, above.

4.10 Comparison to C/C++

The algorithmic language used to express method bodies in DML is an extended subset of ISO C, with some C++ extensions such as `new` and `delete`. The DML-specific statements and expressions are described in Sections 4.11 and 4.12.

DML defines the following additional built-in data types:

int1,...,int64,uint1,...,uint64
: Signed and unsigned specific-width integer types. Widths from 1 to 64 are allowed.

bool
: The generic boolean datatype, consisting of the values `true` and `false`. It is is not an integer type, and the only implicit conversion is to `uint1`

DML also supports the non-standard C extension `typeof(`*expr*`)` operator, as provided by some modern C compilers such as GCC.

Most forms of C statements and expressions can be used in DML, with the exceptions listed below. Some of these limitations may be removed in a later version of DML.

- Local variable declarations must use the keyword `local` or `static`, as in

  ```
  method m() {
      static int call_count = 0;
      local int n = 0;
      local float f;
      ...
  }
  ```

  (only one variable can be introduced per declaration). Local variables must be declared at the beginning of a compound statement. Static variables have the same meaning as in C. For symmetry with C, the keyword `auto` may be used as a synonym for `local`.

- Plain C functions (i.e., not DML methods) can be called using normal function call syntax, as in "`f(x)`", but C functions can not be defined in DML itself; they can either be placed in a C file and be compiled and linked separately, or be placed in a `%footer` or `%header` section in the DML file. In both cases, the function must also be declared as `extern` in the DML source code.

- `return` statements do not take a return value as argument; output parameters of methods must be assigned explicitly.

- Type casts must be written as `cast(`*expr*`, `*type*`)`.

- Comparison operators and logical operators produce results of type `bool`, not integers.

- Conditions in `if`, `for`, `while`, etc. must be proper booleans; e.g., `if (i == 0)` is allowed, and `if (b)` is allowed if `b` is a boolean varable, but `if (i)` is not, if `i` is an integer.

- the `sizeof` operator can only be used on expressions. To take the size of a datatype, the `sizeoftype` operator must be used.

- Comma-expressions are only allowed in the head of `for`-statements, as in

```
for (i = 10, k = 0; i > 0; --i, ++k) ...
```

- `delete` and `throw` can only be used as statements in DML, not as expressions.
- `throw` does not take any argument, and `catch` cannot switch on the type or value of an exception.
- Type declarations do not allow the use of `union`.

4.11 Statements

All ISO C statements are available in DML, and have the same semantics as in C. Like ordinary C expressions, all DML expressions can also be used in expression-statements.

DML adds the following statements:

4.11.1 Delete Statements

`delete` *expr*`;`

Deallocates the memory pointed to by the result of evaluating *expr*. The memory must have been allocated with the `new` operator, and must not have been deallocated previously. Equivalent to `delete` in C++; however, in DML, `delete` can only be used as a statement, not as an expression.

4.11.2 Try Statements

`try` *protected-stmt* `catch` *handle-stmt*

Executes *protected-stmt*; if that completes normally, the whole `try`-statement completes normally. Otherwise, *handle-stmt* is executed. This is similar to exception handling in C++, but in DML 1.0 it is not possible to switch on different kinds of exceptions. This may change in a future version of DML. Note that Simics C-exceptions are not handled. See also `throw`.

4.11.3 Throw Statements

`throw;`

Throws (raises) an exception, which may be caught by a `try`-statement. Exceptions are propagated over method call boundaries. This is similar to `throw` in C++, but in DML 1.0 it is not possible to specify a value to be thrown. This may change in a future version of DML. Furthermore, in DML, `throw` can only be used as a statement, not as an expression.

4.11.4 Call Statements

`call` *method*`(`*e1*`, ... `*eN*`) -> (`*d1*`, ... `*dM*`);`

Calls a DML method with input arguments *e1*, ... *eN* and output destinations *d1*, ... *dM*. The destinations are usually variables, but they can be arbitrary L-values (even bit slices) as long as their types match the method signature.

If the method has no output parameters, the `-> ()` part may be omitted, as in

```
call p(...);
```

which is equivalent to `call p(...) -> ();`.

If the method has no input parameters, the empty pair of parentheses may also be omitted, as in

```
call q -> (...);
```

which is equivalent to `call q() -> (...);`.

A method with neither input nor output parameters may thus be called simply as

```
call me;
```

4.11.5 Inline Statements

`inline` *method*`(`*e1*`, ... `*eN*`) -> (`*d1*`, ... `*dM*`);`

This is equivalent to `call` *method*`(`*e1*`, ... `*eN*`) -> (`*d1*`, ... `*dM*`);` but the code for the called method is expanded at the place of the `inline` call, and may be partly specialized to the values of any input arguments that are constant at DML compile time.

Furthermore, methods that are only intended for inlining may be declared as a form of polymorphic hygienic macros; see Section 4.6.3.

4.11.6 After Statements

`after (`*time*`) call` *method*`;`

The `after` construct sets up an asynchronous event which will perform the specified method call at the given time into the future (in simulated time, measured in seconds) relative to the time when the `after` statement is executed. For example:

```
after (0.1) call $my_callback;
```

This is equivalent to creating a named **event** object with an event-method that performs the specified `call`, and posting that event at the given time; see Section 3.11.

4.11.7 Log Statements

`log` *log-type*`,` *level*`,` *groups*`:` *format-string*`,` *e1*`, ...,` *eN*`;`

Outputs a formatted string to the Simics logging facility. The string following the colon is a normal C `printf` format string, optionally followed by one or more arguments separated by commas. (The format string should not contain the name of the device, or the type of the message, e.g., “error:...”; these things are automatically prefixed.) Either both of *level* and *groups* may be omitted, or only the latter; i.e., if *groups* is specified, then *level* must also be given explicitly.

A Simics user can configure the logging facility to show only specific messages, by matching on the three main properties of each message:

- The *log-type* specifies the general category of the message. The value must be one of the string constants `"info"`, `"error"`, `"undefined"`, `"spec_violation"`, `"target_error"`, or `"unimplemented"`. The most common message types are `info` and `error`; see the *Simics Reference Manual* for further details.
- The *level* specifies at what verbosity level the log messages are displayed. The value must be an integer from 1 to 4; if omitted, the default level is 1. The different levels have the following meaning:
  1. Important messages (displayed at the normal verbosity level)
  2. High-level informative messages
  3. Debugging-level messages
  4. Detailed debugging-level messages, e.g., individual register accesses
- The *groups* argument is an integer whose bit representation is used to select which log groups the message belongs to. If omitted, the default value is 0. The log groups are specific for the device, and must be declared using the `loggroup` device-level declaration. For example, a DML source file containing the declarations

  ```
  loggroup good;
  loggroup bad;
  loggroup ugly;
  ```

  could also contain a log statement such as

  ```
  log "info", 2, (bad | ugly): "...";
  ```

  (note the | bitwise-or operator), which would be displayed if the user chooses to view messages from group `bad` or `ugly`, but not if only group `good` is shown.

Groups allow the user to create arbitrary classifications of log messages, e.g., to indicate things that occur in different states, or in different parts of the device, etc. The two log groups `Register_Read` and `Register_Write` are predefined by DML, and are used by several of the built-in methods.

4.11.8 Assert Statements

`assert` *expr*`;`

Evaluates *expr*. If the result is `true`, the statement has no effect; otherwise, a runtime-error is generated. *expr* must have type `bool`.

4.11.9 Error Statements

`error` *string*`;`

Attempting to compile an `error` statement causes the compiler to generate an error, using the specified string as error message. The string may be omitted; in that case, a default error message is used.

4.11.10 Foreach Statements

`foreach` *identifier* `in (`*expr*`)` *statement*

The `foreach` statement repeats its body (the *statement* part) once for each element in the *list* given by *expr*. The *identifier* is used to refer to the current element within the body. It is *not* used with a `$` prefix.

If *expr* is a list, it is always a DML compile-time constant, and in that case the loop is completely unrolled by the DML compiler. This can be combined with tests on the value of *identifier* within the body, which will be evaluated at compile time.

For example:

```
foreach x in ([3,2,1]) {
    if (x == 1) foo();
    else if (x == 2) bar();
    else if (x == 3) baz();
    else error "out of range";
}
```

would be equivalent to

```
baz();
bar();
```

```
    foo();
```

Only `if` can be used to make such selections; `switch` statements are *not* evaluated at compile time, in DML 1.0. (Also note the use of `error` above to catch any compile-time mistakes.)

4.11.11 Select Statements

`select` *identifier* `in (`*expr*`) where (`*cond-expr*`)` *statement* `else` *default-statement*

The `select` statement is very similar to the `foreach` statement, but executes the *statement* exactly once for the first matching element in the *list* given by *expr*, i.e., for the first element such that *cond-expr* is `true`; or if no element matches, it executes the *default-statement*.

If *expr* is a list, *and* the *cond-expr* only depends on compile-time constants, apart from *identifier*, then the choice will be performed by the DML compiler, and code will only be generated for the selected case.

4.12 Expressions

All ISO C operators are available in DML, except for certain limitations on the comma-operator, the `sizeof` operator, and type casts; see Section 4.10. Operators have the same precedences and semantics as in C

DML adds the following expressions:

4.12.1 The Undefined Constant

`undefined`

The constant `undefined` is an abstract *compile-time only* value, mostly used as a default for parameters that are intended to be overridden. See also the `defined` *expr* test, below.

4.12.2 References

`$`*identifier*

To make a reference to a component of the DML object structure from within a method, the reference (an object-context identifier) must be prefixed by a `$` character; see also Section 4.2. Following the identifier, components may be selected using `.` and `->` as in C. (However, most components in the DML object structure are proper substructures selected with the `.` operator.) For example,

```
    $this.size
    $dev.bank1
    $bank1.r0.hard_reset
```

4.12.3 New Expressions

`new` *type*

Allocates a chunk of memory large enough for a value of the specified type. The result is a pointer to the allocated memory. (The pointer is never null; if allocation should fail, the Simics application will be terminated.)

When the memory is no longer needed, it should be deallocated using a `delete` statement.

4.12.4 Cast Expressions

`cast(`*expr*`,` *type*`)`

Type casts in DML must be written with the above explicit `cast` operator, for syntactical reasons.

Semantically, `cast(`*expr*`,` *type*`)` is equivalent to the C expression `(`*type*`)` *expr*.

4.12.5 Sizeoftype Expressions

`sizeoftype` *type*

The `sizeof` operator in DML 1.0 can only be used on expressions, not on types, for syntactical reasons. To take the size of a datatype, the `sizeoftype` operator must be used, as in

```
    int size = sizeoftype io_memory_interface_t;
```

Semantically, `sizeoftype` *type* is equivalent to the C expression `sizeof (`*type*`)`.

4.12.6 Defined Expressions

`defined` *expr*

This compile-time test evaluates to `false` if *expr* has the value `undefined`, and to `true` otherwise.

4.12.7 List Expressions

`[`*e1*`, ..., `*eN*`]`

A list is a *compile-time only* value, and is an ordered sequence of zero or more compile-time constant values. Lists are in particular used in combination with `foreach` and `select` statements, and are often provided by built-in methods or parameters, such as the `fields` parameter of **register** objects.

4.12.8 Bit Slicing Expressions

expr`[`*e1*`:`*e2*`]`

expr`[`*e1*`:`*e2*`, `*bitorder*`]`

expr`[`*e1*`]`

expr`[`*e1*`, `*bitorder*`]`

If *expr* is of integer type, then the above *bit-slicing* syntax can be used in DML to simplify extracting or updating particular bit fields of the integer. Bit slice syntax can be used both as an expression producing a value, or as the target of an assignment (an L-value), e.g., on the left-hand side of an = operator.

Both *e1* and *e2* must be integers. The syntax *expr*`[`*e1*`]` is a short-hand for *expr*`[`*e1*`:`*e1*`]` (but only evaluating *e1* once).

The *bitorder* part is optional, and selects the bit numbering scheme (the "endianism") used to interpret the values of *e1* and *e2*. If present, it must be an identifier such as `be32`, as in a `bitorder` device-level declaration, and in that case it overrides any global bit order declaration in the program.

The first bit index *e1* always indicates the *most significant bit* of the field, regardless of the bit numbering scheme; cf. Section 4.8.3. If the default little-endian bit numbering is used, the least significant bit of the integer has index zero, and the most significant bit of the integer has index *n* - 1, where *n* is the width of the integer type.

If big-endian bit numbering is used, e.g., due to a "`bitorder be32;`" declaration in the file, or using a specific local bit numbering as in *expr*`[`*e1*`:`*e2*`, be32]`, then the bit corresponding to the little-endian bit number *n* - 1 has index zero, and the least significant bit has the index *n* - 1, where *n* is the specified base width - in this case 32. Note that this is *not* affected by the width of the integer type, in order to avoid subtle errors; e.g., a 16-bit big-endian value could be represented by a 32-bit integer type, but bit slice operations should use `be16` in order to interpret the bit indexes correctly.

4.12.9 Stringify Expressions

# *expr*

Translates the value of *expr* (which must be a compile-time constant) into a string constant. This is similar to the use of # in the C preprocessor, but is performed on the level of compile time values, not tokens. The result is often used with the + string operator.

4.12.10 String Concatenation Expressions

expr1 + *expr2*

If both *expr1* and *expr2* are compile-time string constants, the expression *expr1* + *expr2* concatenates the two strings at compile time. This is often used in combination with the # operator.

Chapter 5

Libraries and Built-ins

5.1 Standard Parameters

This section describes the built-in parameters of the different object types, together with the expected data types of their values.

5.1.1 Object Parameters

The following are the standard parameters defined in all DML objects:

this [reference]
Always refers to the current object, i.e., the nearest enclosing object definition.

parent [reference | undefined]
Always refers to the parent (containing) object. This has the value `undefined` for the **device** object.

desc [string | undefined]
A short summary of the object, for documentation and debugging purposes. This is `undefined` by default, and is expected to be supplied by the programmer. It should preferably be only a few words. (The short-hand syntax for `desc` is described in Section 4.8.) See also `documentation`.

documentation [string | undefined]
A longer documentation string describing the object. The default value is `undefined`. If defined, this is used in automatically generated documentation (mainly for the device and its attributes). If undefined, the `desc` parameter is used instead.

name [string | undefined]
The name of the object, without any qualifiers. This may be `undefined` in implicitly created objects. See also `qname`, below.

qname [string | undefined]
The fully-qualified name of the object, such as `"my_device.bank1.r0"`. This may be `undefined` in implicitly created objects.

objtype [string]
The object type name.

dev [reference]
Normally refers to the device object, i.e., the top-level object definition (regardless of the name used for the device); inherited from parent object.

bank [reference | undefined]
Normally refers to the enclosing bank object; inherited from parent object. Has the value `undefined` for objects not within a bank.

index [integer | undefined]
For objects that are not array elements, this has the default value `undefined`. For further details, see Array Parameters, below.

indexvar [string | undefined]
For objects that are not array elements, this has the default value `undefined`. For further details, see Array Parameters, below.

5.1.2 Array Parameters

array [list of references]
A list of object references to each of the elements of the array.

arraysize [integer]
The number of elements in the array.

index [integer]
For every object that is an array element, this parameter specifies the index of the object in the containing array. The first element of an array always has index 0.

***i* [integer]**
Each array has an *individual index parameter*, to make it possible to refer to both inner and outer indexes when arrays are nested (cf. the `index` parameter, above). The parameter name can be specified in the array declaration, as in "`register regs[j in 0..10]{...}`". The default name is `i`, if the array is defined without an explicit index parameter, as in "`register regs[10]{...}`" (see also `indexvar`, below).

indexvar [string]
This parameter specifies the name of the individual index parameter of the containing array.

5.1.3 Device Parameters

classname [string]
The name of the Simics configuration object class defined by the device model. Defaults to the name of the device object.

banks [list of references]
A list of references to all the **bank** objects of the device object.

register_size [integer | undefined]
The default size (width) in bytes used for **register** objects; inherited by **bank** objects. The default value is `undefined`.

byte_order [string]
The default byte order used when accessing registers wider than a single byte; inherited by **bank** objects. Allowed values are `"little-endian"`, `"big-endian"` and `undefined`. The default value is `"little-endian"`.

log_group [integer | undefined]
An additional log group used by registers when logging register accesses. This may be overridded in each bank and each individual register or register group.

obj [reference]
A special built-in parameter that can be used to provide a Simics API `conf_object_t*` reference to the device object at the C level.

logobj [reference]
A special built-in reference that can be used to provide a Simics API `log_object_t*` reference to the device object at the C level.

5.1.4 Attribute Parameters

allocate_type [string | undefined]
For attributes with simple types such as `bool`, `int32`, `uint32`, `double` and `string`, defining `allocate_type` to the desired type name as a string, e.g., `"uint32"`, will automatically cause storage to be created for the attribute value, and will generate the corresponding get/set methods for the attribute. The default value of this parameter is `undefined`.

type [string | undefined]
A Simics configuration-object attribute type description string, such as `"i"` or `"[s*]"`, specifying the type of the attribute. (See the *Simics Programming Guide* for details.) This is calculated automatically from the `allocate_type` parameter, if that is not `undefined`.

configuration [`"required"` | `"optional"` | `"pseudo"` | `"none"`]
Specifies how Simics treats the attribute. The default value is `"optional"`. A *required* attribute must be initialized to a value when the object is created, while an *optional* attribute can be left undefined. In both cases, the value is saved when a checkpoint is created. For a *pseudo*-attribute, the value is *not* saved when a checkpoint is created (and it is not required to be initialized upon object creation). Setting the value to `"none"` suppresses creation of the attribute; this can sometimes be useful for implicit attributes that inherit the `configuration` parameter from their parent, as e.g., in **register** objects (cf. Section 3.7.2).

persistent [bool]
If this parameter is `true`, the attribute will be treated as persistent, which means that

its value will be saved when using the **save-persistent-state** command. The default value is `false`.

attr_type [string | undefined]
Reserved for internal use. Always has the same value as the `type` parameter.

5.1.5 Bank Parameters

mapped_registers [list of references]
A list of references to all register objects that are mapped into the address space of the bank, sorted by address.

unmapped_registers [list of references]
A list of references to all register objects that are *not* mapped into the address space of the bank, sorted by address.

function [integer | undefined]
The function number of the bank, used for mapping the bank into a Simics memory space. Defaults to `undefined`.

overlapping [bool]
Specifies whether this bank allows accesses that cover more than one register. (This translates to one or more possibly partial accesses to adjacent registers.) Defaults to `false`.

partial [bool]
Specifies whether this bank allows accesses that cover only parts of a register. In a partial access, only the addressed fields of the register are read or written. If there are no explicit fields, the whole register is accessed. Defaults to `false`.

signed [bool]
The default for whether register contents should be treated as signed integers; inherited by **register** objects. The default value is `false`.

allocate [bool]
The default for whether storage for register contents should be allocated in the generated C code; inherited by **register** objects. The default value is `true`.

register_size [integer | undefined]
Inherited from the **device** object; provides the default value for the `size` parameter of **register** objects.

byte_order [string]
Specifies the byte order used when accessing registers wider than a single byte; inherited from **device** objects. Allowed values are `"little-endian"`, `"big-endian"` and `undefined`.

log_group [integer | undefined]
An additional log group used by registers when logging register accesses. This may

be overriddedn in each individual register or register group. The default value is inherited from the **device** object.

miss_bank [reference | undefined]
Can be set to refer to another bank object, to which accesses are forwarded if they cannot be performed in the current bank. Defaults to `undefined`.

miss_bank_offset [integer]
The offset to be added to the address when an access is forwarded through the `miss_bank` parameter. The default value is 0.

5.1.6 Register Parameters

size [integer | undefined]
The default size (width) of the register, in bytes. This parameter can also be specified using the "`size` *n*" short-hand syntax for register objects. The default value is provided by the `register_size` parameter of the enclosing **bank** object.

bitsize [integer]
The size (width) of the register, in bits. This is equivalent to the value of the `size` parameter multiplied by 8, and cannot be overridden.

offset [integer | undefined]
The address offset of the register, in bytes relative to the start address of the bank that contains it. This parameter can also be specified using the "`@` *n*" short-hand syntax for register objects. The default value is `undefined`.

signed [bool]
Specifies whether the register contents should be treated as a signed integer; inherited from **bank** objects.

allocate [bool]
Specifies whether storage for the register contents should be allocated in the generated C code; inherited from **bank** objects.

fields [list of references]
A list of references to all the `field` objects of a register object. There is always at least one field per register.

hard_reset_value [integer]
The value used to initialize the internal state of a register upon hard reset. Defaults to 0.

soft_reset_value [integer]
The value used to initialize the internal state of a register upon soft reset. Defaults to the value of the `hard_reset_value` parameter.

logging [bool]
Specifies whether accesses to the register should be logged. The default is `true`. See also `read_logging` and `write_logging`.

log_group [integer | undefined]
An additional log group used when logging register accesses. The default value is inherited from the **device** object.

read_logging [bool]
Specifies whether read accesses to the register should be logged. Defaults to the value of the `logging` parameter.

write_logging [bool]
Specifies whether write accesses to the register should be logged. Defaults to the value of the `logging` parameter.

configuration [`"required"` | `"optional"` | `"pseudo"`]
Specifies how Simics treats the automatically created attribute corresponding to the register. See Section 5.1.4 for details.

persistent [bool]
Specifies whether the register attribute should be persistent. See Section 5.1.4 for details.

attr_type [string | undefined]
Reserved for internal use. Always has the value `"i"`.

5.1.7 Field Parameters

reg [reference]
Always refers to the containing register object.

explicit [bool]
The value of this parameter is `true` for fields explicitly declared in the DML source code, and is `false` for implicitly created fields.

lsb [integer]
The bit number (*in little-endian bit order*) of the least significant bit of the field, in the containing register; a required parameter. *This is not affected by any `bitorder` declaration.* The preferred way of defining this parameter is to use the "`[`*highbit*`:`*lowbit*`]`" short-hand syntax for field ranges, whose interpretation *is* dependent on the `bitorder` declaration of the file. Care must be taken when referring to this parameter in a big-endian bit numbering system - if possible, put such code in a separate file that uses little-endian bit order interpretation.

msb [integer]
The bit number (*in little-endian bit order*) of the most significant bit of the field, in the containing register; a required parameter. See `lsb` for details.

bitsize [integer]
The size (width) of the field, in bits. This is automatically set from the `lsb` and `msb` parameters and cannot be overridden.

signed [bool]
Specifies whether the field contents should be treated as a signed integer; inherited from the **register** object.

allocate [bool]
Specifies whether storage for the field contents should be allocated in the generated C code; inherited from the **register** object.

hard_reset_value [integer]
The value used to initialize the internal state of the field upon hard reset. Defaults to `undefined`. If the value is `undefined`, the corresponding bits of the value of the `hard_reset_value` parameter of the containing register are used instead.

soft_reset_value [integer]
The value used to initialize the internal state of the field upon soft reset. Defaults to the value of the `hard_reset_value` parameter. If the value is `undefined`, the corresponding bits of the value of the `soft_reset_value` parameter of the containing register are used instead.

5.1.8 Connect Parameters

interfaces [list of references]
A list of references to all the `interface` objects of a connect object.

configuration [`"required"` | `"optional"` | `"pseudo"`]
Specifies how Simics treats the automatically created attribute corresponding to the connect object. See Section 5.1.4 for details.

persistent [bool]
Specifies whether the attribute should be persistent. See Section 5.1.4 for details.

attr_type [string | undefined]
Reserved for internal use.

5.1.9 Interface Parameters

required [bool]
Specifies whether a connected object must implement the interface. If the value is `true` and the object does not implement the interface, a runtime error is generated. The default value is `true`.

c_type [string]
Specifies the C type of the interface (typically a `typedef`:ed name for a struct containing function pointers). Defaults to the name of the interface object with the string `"_interface_t"` appended.

5.1.10 Event Parameters

timebase [`"steps"` | `"cycles"` | `"seconds"` | `"stacked"`]
Specifies the unit for the time value passed to the `post` method. The special unit `"stacked"` means that the value is ignored and the event is posted on the top of the queue. Defaults to `"seconds"`.

5.1.11 Implement Parameters

c_type [string]
Specifies the C type of the implemented interface (typically a `typedef`:ed name for a struct containing function pointers). Defaults to the name of the implement object with the string `"_interface_t"` appended.

5.2 Standard Methods

This section describes the built-in methods of the different object types. Most methods can be redefined by the user, unless otherwise stated.

5.2.1 Array Methods

hard_reset()
Invokes the ***hard_reset*** method on each element of the array.

soft_reset()
Invokes the ***soft_reset*** method on each element of the array.

get_attribute() -> (attr_value_t val)
Not intended to be used directly. Implements reading of an array attribute as a Simics vector, by calling the ***get_attribute*** methods on individual elements of the array.

set_attribute(attr_value_t val) -> (set_error_t err)
Not intended to be used directly. Implements assignment of a Simics vector to an array attribute, by calling the ***set_attribute*** methods on individual elements of the array.

5.2.2 Device Methods

init()
Called when the device object is loaded, but before its configuration-object attributes have been initialized.

post_init()
Called when the device object is loaded, *after* its configuration-object attributes have been initialized.

hard_reset()
Called by Simics when a hard reset is performed on the device. Invokes the ***hard_reset*** method on each **bank** of the device.

soft_reset()
Called by Simics when a soft reset is performed on the device. Invokes the ***soft_reset*** method on each **bank** of the device.

5.2.3 Attribute Methods

get() -> (attr_value_t value)
Returns the value of the attribute.

set(attr_value_t value)
Sets the value of the attribute.

after_set()
Called by ***set_attribute*** just after the ***set*** method has been called. The default implementation does nothing.

get_attribute -> (attr_value_t value)
Not intended to be used directly. Called by Simics for reading the attribute value. Calls the ***get*** method to read the value, and handles any exceptions that may occur.

set_attribute(attr_value_t value) -> (set_error_t err)
Not intended to be used directly. Called by Simics for setting the attribute value. Calls the ***set*** method to set the value, and afterwards calls the ***after_set*** method if ***set*** succeeded. Handles any exceptions that may occur.

5.2.4 Bank Methods

access(generic_transaction_t $*$memop, uint32 offset, uint32 size)
Called when the bank is accessed via the `io_memory` interface. Depending on the `memop` contents, the ***read_access*** or ***write_access*** method is called. For a write access, the method ***get_write_value*** is called to extract the value from the memop. If the access succeeded, the corresponding ***finalize_read_access*** or ***finalize_read_access*** method is called, updating the `memop` before the ***access*** method returns; otherwise, the ***miss_read_access*** or ***miss_write_access*** method, respectively, is called. If this also fails to handle the access, the generic ***miss_access*** method is called, and gets full responsibility for updating the `memop`.

To signal a complete failure to handle the access, the ***access*** method (or any method that it calls, such as ***miss_access***) should raise an exception instead of returning. This will be caught and handled by the `io-memory.dml` library.

read_access(generic_transaction_t $*$memop, uint32 offset, uint32 size) -> (bool success, uint64 value)

Performs a read access by calling the corresponding ***read_access*** method of the addressed register (or registers, for overlapping accesses). Does not update the `memop`. If the access is valid, the `success` output parameter is set to `true`, otherwise to `false`.

read(generic_transaction_t $*$memop, uint32 offset, uint32 size) -> (uint64 value)
Utility method; equivalent to calling ***read_access***, but does not return a success flag.

E.g., can be used to read directly from a bank when it is known that the access will always succeed.

write_access(generic_transaction_t *memop, uint32 offset, uint32 size, uint64 value) -> (bool success)

Performs a write access by calling the corresponding ***write_access*** method of the addressed register (or registers, for overlapping accesses). Does not update the `memop`. If the access is valid, the `success` output parameter is set to `true`, otherwise to `false`.

write(generic_transaction_t *memop, uint32 offset, uint32 size, uint64 value)
Utility method; equivalent to calling ***write_access***, but does not return a success flag. E.g., can be used to write directly to a bank when it is known that the access will always succeed.

miss_read_access(uint32 offset, uint32 size) -> (bool success, uint64 value)
Called from ***access*** upon a missed read access. By default, this function only sets the `success` output parameter to `false` and returns. Provides a simple hook for handling read misses.

miss_write_access(uint32 offset, uint32 size, uint64 value) -> (bool success)
Called from ***access*** upon a missed write access. By default, this function only sets the `success` output parameter to `false` and returns. Provides a simple hook for handling write misses.

miss_access(generic_transaction_t *memop, uint32 offset, uint32 size)
Called from ***access*** when the access could not be handled. This method takes over the responsibility for either updating the `memop` and returning, or raising an exception; see ***access*** for further details.

By default, an info message will be logged. If the parameter `miss_bank` of the bank is not `undefined`, then the access is redirected to the bank referred to by `miss_bank`, adding the value of the parameter `miss_bank_offset` (default 0) to the offset for the access. Otherwise, a specification violation message is logged and an exception is raised.

finalize_read_access(generic_transaction_t *memop, uint64 val)
Called by ***access*** when a read access has succeeded. By default, this method calls ***set_read_value*** to update the `memop`. This method may also be useful to call when overriding ***miss_access***.

finalize_write_access(generic_transaction_t *memop, uint64 val)
Called by ***access*** when a write access has succeeded. By default it has no effect, since the `memop` normally does not need updating after a write; cf. ***finalize_read_access***.

get_write_value(generic_transaction_t *memop) -> (uint64 writeval)
Extracts the value to be written from the `memop`. How this is done depends on the `byte_order` parameter.

set_read_value(generic_transaction_t ∗memop, uint64 value)
Stores the read value in the `memop`. How this is done depends on the `byte_order` parameter.

hard_reset()
Called automatically from the ***hard_reset*** method of the device. Invokes the ***hard_reset*** method on each **register** of the bank.

soft_reset()
Called automatically from the ***soft_reset*** method of the device. Invokes the ***soft_reset*** method on each **register** of the bank.

5.2.5 Register Methods

get() -> (value)
Returns the value of the register, without any other effects. If the register contains fields, this is done by calling the ***get*** method on each **field** of the register and composing the results into a single value.

set(value)
Sets the value of the register, without any other effects. If the register contains fields, this is done by calling the ***set*** method on each **field** of the register, for the corresponding components of the value.

read_access(generic_transaction_t ∗memop, msb1, lsb) -> (value)
Performs a read access by invoking the ***read*** method of the register, or if the register contains fields, by invoking the ***read_access*** method on each of its fields and composing the results into a single value. For a partial access, only affected fields are read. The fields are read in order, starting with the field containing the least significant bits of the register and ending with the field containing the most significant bits of the register. Finally, the ***after_read*** method of the register is called. This method is called from the ***access*** method of the bank.

write_access(generic_transaction_t ∗memop, msb1, lsb, value)
Performs a write access by invoking the ***write*** method of the register, or if the register contains fields, by invoking the ***write_access*** method on each of its fields for the corresponding components of the value. For a partial access, only affected fields are written. The fields are written in order, starting with the field containing the least significant bits of the register and ending with the field containing the most significant bits of the register. Finally, the ***after_write*** method of the register is called. This method is called from the ***access*** method of the bank.

read() -> (value)
Called by ***read_access*** for reading the actual value of the register. This is not used if the register contains fields.

write(value)
Called by ***write_access*** for performing the actual write to the register. This is not used if the register contains fields.

after_read(generic_transaction_t ∗memop)
: Called at the very end of the ***read_access*** method. The default implementation does nothing.

after_write(generic_transaction_t ∗memop)
: Called at the very end of the ***write_access*** method. The default implementation does nothing.

hard_reset_register()
: Called automatically from the ***hard_reset*** method of the bank. Invokes the ***hard_reset*** method of the register, or if the register contains fields, invokes the ***hard_reset*** method on each of its fields. Finally, the ***after_hard_reset*** method of the register is called.

soft_reset_register()
: Called automatically from the ***soft_reset*** method of the bank. Invokes the ***soft_reset*** method of the register, or if the register contains fields, invokes the ***soft_reset*** method on each of its fields. Finally, the ***after_soft_reset*** method of the register is called.

hard_reset()
: Called by ***hard_reset_register*** for performing the actual hard reset. This is not used if the register contains fields.

soft_reset()
: Called by ***hard_reset_register*** for performing the actual soft reset. This is not used if the register contains fields.

after_hard_reset()
: Called at the very end of the ***hard_reset*** method. The default implementation does nothing.

after_soft_reset()
: Called at the very end of the ***soft_reset*** method. The default implementation does nothing.

get_attribute -> (attr_value_t value)
: Not intended to be used directly. Called by Simics for reading the value of the register as an attribute. Calls the ***get*** method to read the value, and handles any exceptions that may occur.

set_attribute(attr_value_t value) -> (set_error_t err)
: Not intended to be used directly. Called by Simics for setting the value of the register as an attribute. Calls the ***set*** method to set the value, and finally calls the ***after_set*** method if ***set*** succeeded. Handles any exceptions that may occur.

after_set()
: Called by ***set_attribute*** just after the ***set*** method has been called, if it succeeded. The default implementation does nothing.

5.2.6 Field Methods

get() -> (value)
Returns the value of the field, without any other effects.

set(value)
Sets the value of the field, without any other effects.

read() -> (value)
Performs an actual read from the field. Called from the ***read_access*** method.

write(value)
Performs an actual write to the field. Called from the ***write_access*** method. The ***write*** functions for the fields in a register are called in a defined order. The field containing the least significant bits of the register are called first, and the field containing the most significant bits are called last. See the documentation for ***write_access*** on a register for more information.

read_access() -> (value)
Usually not used directly. Performs a read access on the field by calling the ***read*** method. Called from the ***read_access*** method of the containing register.

write_access(value)
Usually not used directly. Performs a write access to the field by calling the ***write*** method. Called from the ***write_access*** method of the containing register.

hard_reset()
Performs a hard reset on the field. Called from the ***hard_reset_register*** method of the containing register.

soft_reset()
Performs a soft reset on the field. Called from the ***soft_reset_register*** method of the containing register.

5.2.7 Connect Methods

after_set()
Called at the very end of the ***set_attribute*** method. The default implementation does nothing.

get_attribute -> (attr_value_t value)
Not intended to be used directly. Called by Simics for reading the value of the connect as an attribute.

set_attribute(attr_value_t value) -> (set_error_t err)
Not intended to be used directly. Called by Simics for setting the value of the connect as an attribute. Afterwards calls the ***after_set*** method. Handles any exceptions that may occur.

5.2.8 Interface Methods

Interfaces have no documented standard methods in DML 1.0.

5.2.9 Event Methods

event(void ∗data)
The method called when the event is triggered. The default event method logs a short info message noting that the event occurred.

get_event_info(void ∗data) -> (attr_value_t info)
This method is called once for each pending event instance when saving a checkpoint. It should create an attribute value that can be used to restore the event. The *data* parameter is the user data provided in the ***post*** call. The default implementation always returns a nil value.

set_event_info(attr_value_t info) -> (void ∗data)
This method is used to restore event information when loading a checkpoint. It should use the attribute value to create a user data pointer, as if it had been provided in a ***post***. The default implementation only checks that the checkpointed information is nil.

post(time, void ∗data)
Posts the event on the associated queue of the device, with the specified time (relative to the current time, in the unit specified by the `timebase` parameter), and additional data.

post_on_queue(conf_object_t ∗queue, when, void ∗data)
Like ***post***, but posts the event on the specified queue. This is very rarely needed, since the events should usually always be posted to the queue that the device belongs to, as configured by the *queue* attribute.

remove(void ∗data)
Removes the event from the queue.

posted(void ∗data) -> (flag)
Returns `true` if the event is in the queue, and `false` otherwise.

next(void ∗data) -> (time)
Returns the time to the next occurrence of the event in the queue (relative to the current time, in the unit specified by the `timebase` parameter). If there is no such event in the queue, a negative value is returned.

5.2.10 Implement Methods

Implements have no documented standard methods in DML 1.0.

Appendix A

Known Limitations

The following are known limitations of the `dmlc` compiler at the time of this writing:

Arrays and Simics configuration attributes

Simics configuration attributes are generated for each `register`, `attribute`, and `connect` object in DML. When the object is declared as an array, as in:

```
bank b {
    register r[8] ...;
}
```

the corresponding attribute (*b_r*) of the device is defined to be a *list* attribute. In Python, reading such an attribute will yield a Python list.

However, since the attributes will not be implemented as *indexable*, manipulation of that list will not propagate back into the device, so e.g. writing

```
simics> @conf.b_r[2] = 17
```

will have the same effect as the following:

```
simics> @a = conf.dev.b_r
simics> @print a
[0, 0, 0, 0, 0, 0, 0, 0]
simics> @a[2] = 17
simics> @print a
[0, 0, 17, 0, 0, 0, 0, 0]
simics> @print conf.dev.b_r
[0, 0, 0, 0, 0, 0, 0, 0]
```

in other words, modifying the Python list representation does not change the object that produced the list.

To update one element in the attribute, you need to read the list, modify it, and then write back the whole list to the attribute:

```
simics> @a = conf.dev.b_r
simics> @a[2] = 17
simics> @conf.dev.b_r = a
```

Appendix B

Messages

The following sections list the warnings and error messages from `dmlc`, with some clarifications.

B.1 Warning Messages

The messages are listed in alphabetical order; the corresponding tags are shown within brackets, e.g., `[WNDOC]`.

false assertion [WASSERT]
: The condition of the assertion is always false. (This warning is disabled by default.)

no documentation for '...' [WNDOC]
: No documentation string was specified for the attribute. (This warning is disabled by default.)

possible truncation when shifting [WSHTRUNC]
: The result of the shift operation might not fit in the output type. (This warning is disabled by default.)

shifting away all data [WSHALL]
: The result of the shift operation will always be zero. (This warning is disabled by default.)

the device defines banks, but does not implement io_memory [WNIOMEM]
: A device with register banks typically needs to implement the io_memory interface to be useful.

unused [WUNUSED]
: The object is not referenced anywhere. (This warning is disabled by default.; it typically causes many false warnings.)

B.2 Error Messages

The messages are listed in alphabetical order; the corresponding tags are shown within brackets, e.g., `[ENBOOL]`.

array index out of bounds [EOOB]
The used indexed is outside the defined range.

array range must start at 0 [EZRANGE]
An array index range must start at zero.

array size is less than 1 [EASZR]
An array must have at least one element.

array size is not a constant integer [EASZVAR]
The size of an array must be a constant integer.

attribute type undefined [EATYPE]
Either the `attr_type` or the `type` parameter of the attribute must be specified.

assignment to constant [ECONST]
The lvalue that is assigned to is declared as a `const` and thus can't be assigned to.

bit range of field '...' outside register boundaries [EFBITS]
The bit range of a field can only use bits present in the register.

bit range of field '...' overlaps with field '...' [EBITRO]
The fields of a register must not overlap.

bitslice size of ... bits is not between 0 and 64 [EBSSIZE]
Bit slices cannot be larger than 64 bits.

break is not possible here [EBREAKU]
A `break` statement cannot be used in a `foreach` or `select` statement.

cannot assign to this expression [EASSIGN]
The target of the assignment is not an l-value, and thus cannot be assigned to.

cannot assign to inlined parameter [EASSINL]
The target of the assignment is a method parameter that has been given a constant or undefined value when inlining the method.

cannot define both allocate_type and local 'data' objects [EATTRDATA]
Specifying `allocate_type` and using 'data' declarations in the same attribute object is not allowed.

cannot find file to import [EIMPORT]
The file to imported could not be found. Use the `-I` option to specify additional directories to search for imported files.

cannot use a register with fields as a value [EREGVAL]
When a register has been specified with explicit fields, you have to use the ***get*** and ***set*** methods to access the register as a single value.

cannot use an array as a value [EARRAY]
A whole array cannot be used as a single value.

cannot use variable index in a constant list [EAVAR]
Indexing into constant lists can only be done with constant indexes.

continue is not possible here [ECONTU]
A `continue` statement cannot be used in a `foreach` or `select` statement.

device declarations are not allowed in imported files [EDEVIMP]
Source files that are used with `import` directives may not contain `device` declarations.

duplicate assignment to parameter [EDPARAM]
A parameter may only have a single assignment (apart from any default value).

duplicate bank function number [EDBFUNC]
The device contains two differently-named banks that use the same function number.

duplicate definition of method [EDMETH]
A method has more than one definition in the same scope.

duplicate definition of variable [EDVAR]
A local variable has more than one definition in the same code block.

duplicate method parameter name [EARGD]
All parameter names of a method must be distinct.

file is empty [EEMPTY]
The compilation failed becuse the input file was empty.

forced compilation error in source code [EERSTMT]
The source code contained a statement "`error;`", which forces a compilation error with the above message.

illegal 'after' call [EAFTER]
An illegal `after (..)  call ..` was made. A method called this way may not take any parameters, and it may not be part of an array.

illegal bitslice operation [EBSLICE]
A bitslice operation was attempted on an expression that is not an integer.

illegal bitorder [EBITO]
The specified bit-order is not allowed.

illegal cast to '...' [ECAST]
The cast operation was not allowed. It is illegal to cast to void.

illegal comparison; mismatching types [EILLCOMP]
The values being compared do not have matching types.

illegal function application of . . . [EAPPLY]
The applied value is not a function.

illegal operand to unary '. . . ' [EUNOP]
The operand has the wrong type for the given unary operator.

illegal operands to binary '. . . ' [EBINOP]
One or both of the operands have the wrong type for the given binary operator.

illegal register size [EREGISZ]
The specified register size is not allowed. Possible values are 1-8.

illegal value for parameter [EPARAM]
The parameter is not bound to a legal value.

incompatible array declarations [EAINCOMP]
The array has been declared more than once, in an incompatible way.

incompatible method definitions [EMETH]
The default implementation is overridden by an implementation with different input/output parameters.

incompatible version (. . .) while compiling a . . . device [EVERS]
A device declared to be written in one DML language version tried to import a file writen in an incompatible language version.

invalid expression [EINVALID]
The expression does not produce a proper value.

invalid log type [ELTYPE]
Log-statement type must be one of "info", "error", "undefined", "spec_violation", "target_error", and "unimplemented".

log level must be an integer between 1 and 4 [ELLEV]
Log-statement level must be 1-4.

malformed format string [EFORMAT]
The log-statement format string is malformed.

missing device declaration [EDEVICE]
The main source file given to the DML compiler must contain a `device` declaration.

missing implementation of method [EMMETH]
The method has been declared, but is not given an implementation or a default implementation.

more than one output parameter not allowed in interface methods [EIMPRET]
Methods within an `interface` declaration may have only have zero or one output parameter.

name collision on '...' [ENAMECOLL]
The name is already in use in the same scope.

negative size (...) of bit range for '...' [EBITRN]
The size of the bit range must be positive. Note that the [msb:lsb] syntax requires that the most significant bit (msb) is written to the left of the colon, regardless of the actual bit numbering used.

no assignment to parameter [ENPARAM]
The parameter has been declared, but is not assigned a value or a default value.

no return type [ERETTYPE]
The type of the return value (if any) must be specified for methods that implement interfaces.

no type for {input,output} parameter [ENARGT]
Methods that are called must have data type declarations for all their parameters. (Methods that are only inlined do not need this.)

no value provided for auto-declared parameter '...' [EAUTOPARAM]
A parameter declared as predefined using the 'auto' keyword was not assigned a value at compile time, possibly due to a missing `-D` compiler option.

non-boolean condition [ENBOOL]
Conditions must be properly boolean expressions; e.g., "`if (i == 0)`" is allowed, but "`if (i)`" is not, if `i` is an integer.

non-constant element in list [ECLST]
Lists may only contain constants.

non-constant expression [ENCONST]
A constant expression was expected.

non-constant parameter, or circular parameter dependencies [EVARPARAM]
The value assigned to the parameter is not a well-defined constant.

not a list [ENLST]
A list was expected.

not a method [ENMETH]
A method name was excpected.

not a pointer [ENOPTR]
A pointer value was expected.

not a value [ENVAL]
Only some objects can be used as values directly. An attribute can only be accessed directly as a value if it has been declared using the `allocate_type` parameter.

nothing to break from [EBREAK]
A `break` statement can only be used inside a loop or switch construct.

nothing to continue [ECONT]
A `continue` statement can only be used inside a loop construct.

object is not allocated at run-time [ENALLOC]
An object which is not allocated at run-time cannot be referenced as a run-time value.

overlapping registers [EREGOL]
The registers are mapped to overlapping address ranges.

passing const reference for nonconst parameter '...' in function call [ECONSTP]
C function called with a pointer to a constant value for a parameter declared without const in the prototype.

recursive inline [ERECUR]
Methods may not be inlined recursively.

redefining auto parameter [EREAUTOPARAM]
Built-in parameters cannot be redefined by the user.

reference to unknown object [EREF]
The referenced object has not been declared.

right-hand side operand of '...' is zero [EDIVZ]
The right-hand side of the given / or % operator is always zero.

shift with negative shift count [ESHNEG]
The right-hand side operand to a shift operator must not be negative.

syntax error [ESYNTAX]
The code is malformed.

this object is not allowed here [ENALLOW]
Many object types have limitations on the contexts in which they may appear.

trying to get a member of a non-struct [ENOSTUCT]
The left-hand side operand of the . operator is not of struct type.

trying to index something that isn't an array [ENARRAY]
Indexing can only be applied to arrays, integers (bit-slicing), and lists.

uncaught exception [EBADFAIL]
An exception is thrown in a context where it will not be caught.

undefined array size [EASZUNDEF]
The size of an array must be defined somewhere in the program.

undefined index value [EIDX]
The index value is not well-defined.

undefined register size [EREGNSZ]
All registers must have a specified constant size.

undefined value [EUNDEF]
Caused by an attempt to generate code for an expression that contains the `undefined` value.

unknown identifier [EIDENT]
The identifier has not been declared anywhere.

unknown interface type [EIFTYPE]
The interface datatype is unknown.

unknown struct member [EMEMBER]
Attempt to access a nonexisting member of a struct.

unknown template [ENTMPL]
The template has not been defined.

unknown type [ETYPE]
The data type is not defined in the DML code.

wrong number of arguments for format string [EFMTARGN]
The log-statement has too few or too many arguments for the given format string.

wrong number of {input,output} arguments [EARG]
The number of input/output arguments given in the call differs from the method definition.

wrong type in assignment [EASTYPE]
The target of an assignment can't store the value given in the source, because they are of different types.

wrong type for argument *n* of format string ... [EFMTARGT]
Argument type mismatch in a log-statement format string.

wrong type for {input,output} parameter '...' when {calling,inlining} [EARGT]
The data type of the argument value given for the mentioned method parameter differs from the method definition.

wrong type for parameter '...' in function call [EPTYPE]
The data type of the argument value given for the mentioned C function parameter differs from the function prototype.

Appendix C

DML Language Changes

C.1 Changes from DML 0.9 to DML 1.0

DML 0.9 was the version used internally at Virtutech before it was release externally as DML 1.0.

- More C-like syntax:
  - Multi-line `/*...*/`-comments are allowed. Comments do not nest.
  - Identifiers cannot contain hyphens (`-`); use underscores instead.
  - `for`-statements have changed to `for (...;...;...) ...` as in C. Comma-separated expressions are allowed in the first and third sections of the for-statement head. Declarations in the first section of the head are not allowed, yet.
  - `verbatim <typedecl>;` changed to `extern <typedecl>;`.
  - C function types are handled.
  - The keyword `exit` has been changed to `return`. `exit` is no longer a reserved word.
  - The keyword `fail` has been changed to `throw`, as in C++. `fail` is no longer a reserved word.
  - The keyword `free` has been changed to `delete`, as in C++. `free` is no longer a reserved word.
  - The keyword `except` has been changed to `catch`, as in C++. `except` is no longer a reserved word.
  - Pre/post-increment/decrement operators `++`, `--` have been added.
  - The shift-assignment operators <<= and >>= have been added.
  - Assignments can be used as expressions, not only as statements. (This is mainly of importance in the head of for-statements.)
  - `switch`-statements, labels, and `goto`-statements have been added.
  - All ANSI C and C99 keywords are reserved in DML.
  - The C++ keywords `class`, `namespace`, `private`, `protected`, `public`, `this`, `using`, and `virtual` are reserved in DML for future use.

  - The keyword `auto` may be used as a synonym for `local`.

- The keyword `bitslice-syntax` has changed to `bitorder`, and the colon following the keyword in such declarations has been dropped.

  The keyword `log-group` has changed to `loggroup`.

- Import declarations on the form `import <identifier>;` have been disallowed. The file name must be quoted, and must include the extension, as in e.g. `import "io-memory.dml";`.

- The syntax for `log`-statements has changed to the form `log "class", log_level, log_group : "format-string", arg1, ..., argN;`, using a string instead of an identifier to specify the class, and a colon before the format string. The log level is optional, and defaults to 1. The log class names must use underscores, as in "spec_ violation", not "spec-violation" as previously.

- The syntax for `field`-objects has changed to require `[...]` brackets around the range, as in `field nyb0 [3:0];`. It is also possible to specify a 1-bit field using a single number without a colon, as in `field bit7 [7];`

- A simple syntax has been added for registering a callback to be called later in the simulation. Instead of explicitly creating an `event` object and specifying parameters and methods in it, an `after (...) call ...` statement can be used; e.g. `after (0.01) call $mymethod(...);`.

- The standard library file "io-mapped.ddl" has been renamed to "io-memory.dml".

- The standard library file "builtin.ddl" has been renamed to "utility.dml".

- The standard library file "hardwire.ddl" has been renamed to "dml-builtins.dml" (note however that the user has never needed to import this file explicitly; it is done automatically by dmlc).

- The device method `finalize` has been renamed to `post_init`.

  The syntax `export method ...` has been changed to `method extern ...`; `export` is no longer a reserved word.

Appendix D

Grammar

The following section shows the grammar for the DML 1.0 language. It has been automatically generated from the parser grammar used by `dmlc`.

D.1 Formal Grammar

dml →
  device *objident* "**;**" *syntax_modifiers* *device_statements*
  | *syntax_modifiers* *device_statements*

syntax_modifiers →
  <empty>
  | *syntax_modifiers* *syntax_modifier*

syntax_modifier →
  bitorder *ident* "**;**"

device_statements →
  device_statements *device_statement*
  | <empty>

device_statement →
  object_statement
  | *toplevel*
  | *import*

object →
  bank *maybe_objident* *object_spec*
  | **register** *objident* *sizespec* *offsetspec* *istemplate* *object_spec*
  | **register** *objident* "**[**" *arraydef* "**]**" *sizespec* *offsetspec* *istemplate* *object_spec*
  | **field** *objident* *bitrange* *istemplate* *object_spec*
  | **field** *objident* *istemplate* *object_spec*
  | **data** *cdecl* "**;**"
  | **connect** *objident* *object_spec*
  | **interface** *objident* *object_spec*

| **attribute** *objident* *object_spec*
| **event** *objident* *object_spec*
| **group** *objident* *object_spec*
| **implement** *objident* *object_spec*
| **attribute** *objident* "**[**" *arraydef* "**]**" *object_spec*
| **group** *objident* "**[**" *arraydef* "**]**" *object_spec*
| **connect** *objident* "**[**" *arraydef* "**]**" *object_spec*

method →
method *objident* *method_params* *method_def*
| **method extern** *objident* *method_params* *method_def*

arraydef →
expression
| *ident* **in** *expression* "**..**" *expression*

toplevel →
template *objident* *object_spec*
| **header** "%{ ... %}"
| **footer** "%{ ... %}"
| **loggroup** *ident* "**;**"
| **constant** *ident* "**=**" *expression* "**;**"
| **extern** *cdecl_or_ident* "**;**"
| **typedef** *cdecl* "**;**"
| **struct** *ident* "{" *struct_decls* "}"

istemplate_stmt →
is *objident* "**;**"

import →
import *string-literal* "**;**"

object_desc →
string-literal
| <empty>

object_spec →
object_desc "**;**"
| *object_desc* "{" *object_statements* "}"

object_statements →
object_statements *object_statement*
| <empty>

object_statement →
object
| *parameter*
| *method*
| *istemplate_stmt*

parameter →
  parameter *objident* *paramspec*

paramspec →
  *"***;***"*
  | *"***=***"* *expression* *"***;***"*
  | **default** *expression* *"***;***"*
  | **auto** *"***;***"*

method_params →
  <empty>
  | *"***(***"* *cdecl_or_ident_list* *"***)***"*
  | *"***->***"* *"***(***"* *cdecl_or_ident_list* *"***)***"*
  | *"***(***"* *cdecl_or_ident_list* *"***)***"* *"***->***"* *"***(***"* *cdecl_or_ident_list* *"***)***"*

returnargs →
  <empty>
  | *"***->***"* *"***(***"* *expression_list* *"***)***"*

method_def →
  compound_statement
  | **default** *compound_statement*

istemplate →
  <empty>
  | **is** *"***(***"* *objident_list* *"***)***"*

sizespec →
  size *expression*
  | <empty>

offsetspec →
  *"***@***"* *expression*
  | <empty>

bitrange →
  *"***[***"* *expression* *"***]***"*
  | *"***[***"* *expression* *"***:***"* *expression* *"***]***"*

cdecl_or_ident →
  cdecl

cdecl →
  typeident *cdecl2*
  | *struct* *cdecl2*
  | *typeof* *cdecl2*
  | **const** *typeident* *cdecl2*

cdecl2 →
cdecl3
| **const** *cdecl2*
| "∗" *cdecl2*
| **vect** *cdecl2*

cdecl3 →
typeident
| <empty>
| *cdecl3* "**[**" *expression* "**]**"
| *cdecl3* "**(**" *cdecl_list* "**)**"
| "**(**" *cdecl2* "**)**"

cdecl_list →
<empty>
| *cdecl_list2*

cdecl_list2 →
cdecl
| **ELLIPSIS**
| *cdecl_list2* "**,**" *cdecl*
| *cdecl_list2* "**,**" **ELLIPSIS**

cdecl_or_ident_list →
<empty>
| *cdecl_or_ident_list2*

cdecl_or_ident_list2 →
cdecl_or_ident
| *cdecl_or_ident_list2* "**,**" *cdecl_or_ident*

typeof →
typeof *expression*

struct →
struct "**{**" *struct_decls* "**}**"

struct_decls →
struct_decls *cdecl* "**;**"
| <empty>

ctypedecl →
"**(**" *ctypedecl* "**)**"
| **typeof** *expression*
| *typeident* *ctypedecl_pointer*

ctypedecl_pointer →
<empty>
| "∗" *ctypedecl_pointer*

typeident $\rightarrow$
ident
| **char**
| **double**
| **float**
| **int**
| **long**
| **short**
| **signed**
| **unsigned**
| **void**

comma_expression $\rightarrow$
expression
| *comma_expression* “**,**” *expression*

expression $\rightarrow$
expression “**=**” *expression*
| *expression* “**+=**” *expression*
| *expression* “**-=**” *expression*
| *expression* “**$*$=**” *expression*
| *expression* “**/=**” *expression*
| *expression* “**%=**” *expression*
| *expression* “**|=**” *expression*
| *expression* “**&=**” *expression*
| *expression* “**^=**” *expression*
| *expression* “**<<=**” *expression*
| *expression* “**>>=**” *expression*
| *expression* “**?**” *expression* “**:**” *expression*
| *expression* “**+**” *expression*
| *expression* “**-**” *expression*
| *expression* “**$*$**” *expression*
| *expression* “**/**” *expression*
| *expression* “**%**” *expression*
| *expression* “**<<**” *expression*
| *expression* “**>>**” *expression*
| *expression* “**==**” *expression*
| *expression* “**!=**” *expression*
| *expression* “**<**” *expression*
| *expression* “**>**” *expression*
| *expression* “**<=**” *expression*
| *expression* “**>=**” *expression*
| *expression* “**||**” *expression*
| *expression* “**&&**” *expression*
| *expression* “**|**” *expression*
| *expression* “**^**” *expression*

| *expression* “**&**” *expression*
| **cast** “**(**” *expression* “**,**” *ctypedecl* “**)**”
| **sizeof** *expression*
| “**-**” *expression*
| “**+**” *expression*
| “**!**” *expression*
| “**~**” *expression*
| “**&**” *expression*
| “$*$” *expression*
| **defined** *expression*
| “**#**” *expression*
| “**++**” *expression*
| “**--**” *expression*
| *expression* “**++**”
| *expression* “**--**”
| *expression* “**(**” *expression_list* “**)**”
| *integer-literal*
| *hex-literal*
| *binary-literal*
| *float-literal*
| *string-literal*
| **undefined**
| “**$**” *objident*
| *ident*
| *expression* “**.**” *objident*
| *expression* “**->**” *objident*
| **sizeoftype** *ctypedecl*
| **new** *ctypedecl*
| “**(**” *expression* “**)**”
| “**[**” *expression_list* “**]**”
| *expression* “**[**” *expression* *endianflag* “**]**”
| *expression* “**[**” *expression* “**:**” *expression* *endianflag* “**]**”

endianflag →
“**,**” *identifier*
| <empty>

expression_opt →
expression
| <empty>

comma_expression_opt →
comma_expression
| <empty>

expression_list →
<empty>

| *expression*
| *expression* “**,**” *expression_list*

statement →
compound_statement
| “**;**”
| *expression* “**;**”
| **if** “**(**” *expression* “**)**” *statement*
| **if** “**(**” *expression* “**)**” *statement* **else** *statement*
| **while** “**(**” *expression* “**)**” *statement*
| **do** *statement* **while** “**(**” *expression* “**)**”
| **for** “**(**” *comma_expression_opt* “**;**” *expression_opt* “**;**” *comma_expression_opt* “**)**” *statement*
| **switch** “**(**” *expression* “**)**” *statement*
| **delete** *expression* “**;**”
| **try** *statement* **catch** *statement*
| **after** “**(**” *expression* “**)**” **call** *expression* “**;**”
| **call** *expression* *returnargs* “**;**”
| **inline** *expression* *returnargs* “**;**”
| **assert** *expression* “**;**”
| **log** *string-literal* “**,**” *expression* “**,**” *expression* “**:**” *string-literal* *log_args* “**;**”
| **log** *string-literal* “**,**” *expression* “**:**” *string-literal* *log_args* “**;**”
| **log** *string-literal* “**:**” *string-literal* *log_args* “**;**”
| **select** *ident* **in** “**(**” *expression* “**)**” **where** “**(**” *expression* “**)**” *statement* **else** *statement*
| **foreach** *ident* **in** “**(**” *expression* “**)**” *statement*
| *ident* “**:**” *statement*
| **case** *expression* “**:**” *statement*
| **default** “**:**” *statement*
| **goto** *ident* “**;**”
| **break** “**;**”
| **continue** “**;**”
| **throw** “**;**”
| **return** “**;**”
| **error** “**;**”
| **error** *string-literal* “**;**”

log_args →
<empty>
| *log_args* “**,**” *expression*

compound_statement →
“{” *locals* *statement_list* “}”
| “{” *statement_list* “}”
| “{” *locals* “}”
| “{” “}”

statement_list →
 statement
 | *statement_list* *statement*

locals →
 local
 | *locals* *local*

local_keyword →
 local
 | **auto**

local →
 local_keyword *cdecl* "**;**"
 | **static** *cdecl* "**;**"
 | *local_keyword* *cdecl* "**=**" *expression* "**;**"
 | **static** *cdecl* "**=**" *expression* "**;**"

objident_list →
 objident
 | *objident_list* "**,**" *objident*

maybe_objident →
 objident
 | <empty>

objident →
 ident
 | **this**
 | **register**
 | **signed**
 | **unsigned**

ident →
 identifier
 | **attribute**
 | **bank**
 | **bitorder**
 | **connect**
 | **constant**
 | **data**
 | **device**
 | **event**
 | **field**
 | **footer**
 | **group**
 | **header**
 | **implement**

| **import**
| **interface**
| **loggroup**
| **method**
| **size**
| **class**
| **enum**
| **namespace**
| **private**
| **protected**
| **public**
| **restrict**
| **union**
| **using**
| **virtual**
| **volatile**

Index