virtutech

Device Modeling Guidelines

Simics Version	3.0
Revision	1406
Date	2008-02-19

VIRTUTECH CONFIDENTIAL

Contents

Chapter 1

About Simics Documentation

1.1 Conventions

Let us take a quick look at the conventions used throughout the Simics documentation. Scripts, screen dumps and code fragments are presented in a `monospace` font. In screen dumps, user input is always presented in bold font, as in:

```
Welcome to the Simics prompt
simics> this is something that you should type
```

Sometimes, artificial line breaks may be introduced to prevent the text from being too wide. When such a break occurs, it is indicated by a small arrow pointing down, showing that the interrupted text continues on the next line:

```
This is an artificial ↵
line break that shouldn't be there.
```

The directory where Simics is installed is referred to as `[simics]`, for example when mentioning the `[simics]/README` file. In the same way, the shortcut `[workspace]` is used to point at the user's workspace directory.

1.2 Simics Guides and Manuals

Simics comes with several guides and manuals, which will be briefly described here. All documentation can be found in `[simics]/doc` as Windows Help files (on Windows), HTML files (on Unix) and PDF files (on both platforms). The new Eclipse-based interface also includes Simics documentation in its own help system.

Simics Installation Guide for Unix and for Windows

These guides describe how to install Simics and provide a short description of an installed Simics package. They also cover the additional steps needed for certain features of Simics to work (connection to real network, building new Simics modules, . . .).

Simics User Guide for Unix and for Windows

These guides focus on getting a new user up to speed with Simics, providing information on Simics features such as debugging, profiling, networks, machine configuration and scripting.

Simics Eclipse User Guide

This is an alternative User Guide describing Simics and its new Eclipse-based graphical user interface.

Simics Target Guides

These guides provide more specific information on the different architectures simulated by Simics and the example machines that are provided. They explain how the machine configurations are built and how they can be changed, as well as how to install new operating systems. They also list potential limitations of the models.

Simics Programming Guide

This guide explains how to extend Simics by creating new devices and new commands. It gives a broad overview of how to work with modules and how to develop new classes and objects that fit in the Simics environment. It is only available when the DML add-on package has been installed.

DML Tutorial

This tutorial will give you a gentle and practical introduction to the Device Modeling Language (DML), guiding you through the creation of a simple device. It is only available when the DML add-on package has been installed.

DML Reference Manual

This manual provides a complete reference of DML used for developing new devices with Simics. It is only available when the DML add-on package has been installed.

Simics Reference Manual

This manual provides complete information on all commands, modules, classes and haps implemented by Simics as well as the functions and data types defined in the Simics API.

Simics Micro-Architectural Interface

This guide describes the cycle-accurate extensions of Simics (Micro-Architecture Interface or MAI) and provides information on how to write your own processor timing models. It is only available when the DML add-on package has been installed.

RELEASENOTES and LIMITATIONS files

These files are located in Simics's main directory (i.e., `[simics]`). They list limitations, changes and improvements on a per-version basis. They are the best source of information on new functionalities and specific bug fixes.

Simics Technical FAQ

This document is available on the Virtutech website at `http://www.simics.net/support`. It answers many questions that come up regularly on the support forums.

Simics Support Forum

The Simics Support Forum is the main support tool for Simics. You can access it at `http://www.simics.net`.

Other Interesting Documents

Simics uses Python as its main script language. A Python tutorial is available at `http://www.python.org/doc/2.4/tut/tut.html`. The complete Python documentation is located at `http://www.python.org/doc/2.4/`.

Chapter 2

Introduction

This document is a high-level guide to modeling devices in Simics, focusing on issues like performance, abstraction and levels of detail. It collects and summarizes experience from within Virtutech. The main purpose is to help beginners write device models that are accurate enough while still being efficient.

We define a *device* to be a piece of real hardware whose behavior we want to simulate in Simics, but which is not a processor. A *device model* is a Simics module which implements the behavior of the device; to *model* something is to implement it in software for simulation.

2.1 Prerequisites

To create your own device models, you need the following:

- General programming skills, and some knowledge about hardware, low-level programming and device drivers.
- Some general ideas about device modeling (this document).
- A build environment for Simics modules, described in the *Simics Programming Guide*.
- Documentation on how to use your chosen implementation language (typically DML) to write Simics device models (see the *DML Reference Manual*, the *DML Tutorial*, and the *Simics Programming Guide*).
- Documentation of the Simics API, found in the *Simics Reference Manual*.

The *Simics User Guide* contains an overview of the documentation that is available for Simics.

2.2 Simics Basics

Simics is a full-system simulator. Its purpose is to replace real hardware in testing, development, research, etc. In order to be useful, Simics must be able to run realistic workloads. This requires a careful balancing of accuracy and efficiency.

Technically, Simics can be classified as an event-driven simulator. It models the world as a series of disjoint events, where each event occurs at some specific time. Events can query and modify the simulated state, and post new events. This approach is very efficient, since the simulator does nothing at all when there are no events, and can be made arbitrarily accurate. The alternative, clock-driven simulation, can never be made as fast.

Simics simulates a CPU at the instruction set level, the lowest level that is readily visible to software. This lets it run unmodified binary code for the complete software stack of a system. To make this work, the simulator provides bit-exact functional accuracy at this level. Somewhat less accuracy is normally provided in the time domain. However, time accuracy can be selectively improved if needed, at the cost of sacrificing some efficiency.

2.3 Transaction-Oriented vs. Bit-Level Device Models

The defining difference between full-system simulation and traditional instruction set simulation is the modeling of devices. Device models are necessary in order to get meaningful software to run on the simulator: timers, network interfaces, PCI bridges, SCSI interfaces, graphics devices, serial ports - there is a large number of different devices in a computer system that are visible to the software, and therefore need to be modeled.

Device models can be either transaction-oriented or bit-level. A transaction-oriented model handles each interaction (typically, a read or write access to the interface registers of a device) as an atomic operation: the device is presented with a request, computes the reply, and returns it in a single function call. This synchronous model is very efficient.

A bit-level model instead models the actual bit patterns traveling over buses and pins in the computer. Instead of a synchronous transaction, each phase of a transaction is played out as a unidirectional message. The device model will read the address and data lines, wait for some cycles, and then put its reply on the bus connecting it to the CPU. This is an asynchronous style that results in much lower performance, since the simulation needs to switch contexts more often.

Of course, it is possible to create hybrids between the two extremes described above. One quite common approach is to let most of a system work at the transaction level, modeling only a small part of the system at the asynchronous bit-level. This offers an efficient way to drive a detailed model of a device with real traffic from a full system.

Most of the guidelines in this document assume that a (mainly) transaction-oriented device model is being developed.

Chapter 3

General Concepts

There are three main goals when writing a device for Simics: accuracy, efficiency, and simplicity.

Every device should be *accurate* enough that the software that executes within Simics does not notice that it is running in a simulator rather than on real hardware. Note that the accuracy is thus defined relative to some particular piece of software: the device is accurate enough if the given software cannot tell the difference between the model and reality.

We want our simulations to run as fast as possible. Devices that are used frequently should therefore be as *efficient* as possible. General programming principles of course apply, but it is even more important to understand how the device is being accessed during simulation, in order to optimize its performance.

Simplicity is always an important goal in software development, since it leads to reduced development time, reduced bug count, and easier debugging. Device modeling is no exception to this rule. Often, simplicity and efficiency go hand in hand, so that a simpler device model is also faster.

There is a conflict between the accuracy and efficiency goals: more accuracy typically leads to reduced efficiency. To maximize this trade-off, one should strive to model as little as possible of the real behavior of the device. The process of simplifying the behavior of the device in the model compared to the real world is called *abstraction*. Later sections will show different kinds of abstraction that can be used to make your device model both simpler and more efficient. In order to know how much abstraction is permissible, it is essential to look at the behavior of the particular software that will access the device, typically a device driver in the operating system.

Of course, there are situations where additional requirements affect the level of detail in your model. For most models it is only necessary that the software runs correctly on top of them, but the models are not of further interest in themselves. However, if you for example want to study disk accesses, you may need to build a more detailed model of the disks than you would in order to simply run a system that contains a disk.

3.1 Optimization

To increase the performance of your model, you should optimize for the common case, even if it means that some uncommon cases become much slower. Also, it is usually more

important to reduce *how often* the device model is invoked, than reducing the time spent in each invocation.

The simulator framework can help you avoid unnecessary model invocations. For example, rather than having the device model poll a memory location repeatedly, you can set a breakpoint which is triggered only when the value at that location changes.

Chapter 4

Methodology

The typical development methodology can be summed up in the phrase *the device model is the inverse of the driver*. In short, it works as follows: Starting with a skeleton model, boot up the driver in Simics and use the inspection facilities to see what device registers it accesses first. Model those, and if necessary the subsystem behind them, using the hardware documentation for reference. With the additions in place, run the driver again to find the next part of the device that needs modeling. This is repeated until no more unaccounted-for references are found, at which time the device is declared finished.

The skeleton model needs to be detailed enough to allow logging of the activities of the driver with respect to the device. For example, all interface registers may be modeled, but their semantics may be left unimplemented.

Running the software will also help the developer understand what modes of functioning will or will not be used in the device model, such as commands or setup options that are never used.

If the hardware documentation is not available, it is often possible to use a well documented open source device driver for reference when implementing the semantics of the device.

See section 8 for more on how to test device models during development.

Chapter 5

Abstraction of Hardware

A Simics device model should be transparent to the software, so that the software does not notice the difference between the model and the real device. This sets the basic level of detail for the model: anything that is not visible to the software (directly or indirectly) may be ignored (if not of interest for other reasons). It is not necessary to model things like the current status of the signals at the interface pins, hidden internal hardware registers, etc., unless these can be accessed by the software. The model provides the functional behavior of the device as seen by the software.

Since Simics usually controls both parties in any transactions at the hardware level, hardware-hardware interfaces can usually be heavily simplified. For example, transactions on a memory bus or device interconnect are usually modeled as atomic, rather than composed of individual steps where buses are arbitrated, addresses written, data read, etc.

Another example is the standard Simics cache model: the caches hold no data, they just keep track of what items are in the cache, and inform the CPU model of the delay associated with each memory access. The actual data is acquired from the model of main memory, regardless of what cache level the item was found in.

For network interfaces, the device models can often abstract away internal buffers that store data during transmission or reception: each packet is sent immediately upon request, and other parts of the simulation make sure it is received with the proper delay.

Frequently repeated actions should be avoided in all device models. In the network case, the hardware-level protocols should as far as possible abstract away things like idle-traffic (packets sent to ascertain that the link and/or particular devices are still there). This is usually OK, since the simulator controls both the link and the devices connected to it, and can check by other means that they are still alive.

Sometimes there are requirements on what state should be inspectable in the device model, which can affect the level of detail. For example, most memory systems have a number of small internal buffers that are used for temporary storage during memory transactions. If these are to be available for inspection, they must be modeled somehow. Sometimes it may be necessary to provide two models with different levels of detail, allowing the user to select the appropriate trade-off between accuracy and efficiency.

Performance-critical transactions between devices can sometimes be abstracted away by merging the devices into a single model. As an example, consider a large low-latency token ring network, represented as one device model per node. Simulating the message transfer

as individual hops around the ring will require a lot of expensive network synchronization for the processes involved, which will make the simulation run slowly. Increasing the latency on the links speeds up the simulation, but slows down the simulated network instead, which could break the protocols because of timeouts. If the accuracy requirements permit, the performance of such a system can be greatly improved by modeling it at a much more abstract level: The ring is treated as a single device, which means the forwarding of messages around the ring can be ignored, allowing the model to route packages directly to the intended receiver instead.

Chapter 6

Abstraction of Hardware-Software Interfaces

Abstracting the hardware-software interface is more tricky, since the simulator does not control the software the same way it controls the hardware. However, since the device accuracy is defined relative to the driver software that accesses it, there are quite a few things that can be done once the driver and the usage pattern is known.

The obvious thing to do is to abstract away all parts of the interface that are never accessed by the driver. Control registers not used by the driver, states that are never entered, and behaviors that are never triggered can well be left unimplemented. For debugging reasons one should make sure to indicate to the user whenever accesses to unimplemented parts of the model occur. See section 9.3 for more on error reporting and user interface issues.

Another area where the hardware-software interface can be abstracted is timing. This is covered in more detail in section 7.

6.1 Fewer States

The device model can be simplified by reducing the number of states that the device can be in. Look for states that, from the software's perspective, are only there to optimize performance. Here are some examples:

- SCSI devices can become disconnected, and when disconnected they behave differently than when connected. In most cases, we can ignore the disconnected state, and let the device be connected all the time.
- Some network devices can be put in "early interrupt" mode, where interrupts are sent some time before a buffer fills up, to give the driver more time to handle the data. It may be possible to ignore the early interrupt state, if the driver can handle the load anyway. In this case, it is a good idea to warn the user when the driver tries to put the device in early interrupt mode.

- Some devices cache data to speed things up, and provide some means to query whether the cached value is valid. The simplest way to model this is to ignore this state, and always report "no, the cache is not valid".

6.2 Avoid Modeling Error States

The device model is typically expected to be accurate under normal working conditions. Error states (those that do not occur under normal conditions) should never be entered. Most errors are hardware induced; e.g., parity errors, failure to read firmware ROM data, etc. These will never occur in Simics, because the virtual hardware is controlled by the simulator. The device model is simplified by not having to represent the error states.

Sometimes, though, the errors are the interesting parts of the model. If the model is to be used in developing the error handling in a device driver, the error states need to be modeled in more detail (and some means of triggering the errors must be added). Fault-injection in simulated networks is another example.

6.3 Approximate Performance Counters and Hardwire Diagnostics

Performance meters and diagnostic functions are usually not accessed during normal operation, and should therefore be abstracted away. If they are ever read, diagnostic registers should normally be hardwired to report "everything OK" without bothering to access any internal state. If performance meters are accessed, it may be necessary to provide approximate values which are computed only when requested.

If you find that the driver reads values from, e.g., a JTAG port, you can look at the driver source code and try to figure out what values it expects to find (look at what it compares the data to), and make the model supply some values that are acceptable.

Sometimes it is necessary to model a bit more. One particular architecture provides interfaces to access parity bits and checksums in the caches. In its boot sequence, the OS performs a self-test on the caches by writing flawed parity bits and checking that the cache handles them gracefully (reporting error or auto-correcting the data). The model of this cache thus needs to simulate the parity bits. To increase performance, however, it is sufficient to simulate this only on cache lines where the parity bits have been accessed.

When approximate or invented values are being returned from the model, it is good practice to issue a warning to the user, and/or print a message to the appropriate log file. See section 9.3 for more on this.

Chapter 7

Abstraction of Time

Abstraction of time is one of the most important issues in device modeling. Simics already abstracts away some details of the timing of instruction execution and scheduling in the processor, to achieve better performance; see the *Simics User Guide*, "Advanced Simics Usage: Understanding Simics Timing", for more information.

Modern device drivers are often written to be largely independent of the detailed timing of the hardware. This means the device model can alter the timing behavior quite a bit without the driver noticing.

7.1 Provide Reaction Immediately

Where possible, the device model should react immediately to stimuli, even though the real device would take some time to finish the effect. This improves efficiency, because the model is invoked fewer times, and simplifies the implementation since there is no need to store transaction state for later or insert things into the event queues.

For example, a model of a network device can send a packet immediately upon request, reading the content directly from main memory rather than, e.g., initiating a DMA transfer to an internal FIFO, waiting for it to fill up, etc. Another example is an address/data register pair, where the real device requires that the data access must not occur within a specified time from the address write. The model does not need to check for this condition, since the driver will always wait long enough before attempting to read or write the data.

It is often useful to have a simple configurable delay for hardware events that take time. Sometimes the software is sensitive to things that occur too quickly (e.g., immediately) in the model compared to the real world. Adjusting a delay attribute is a simple solution for such problems.

Often, hardware reports the completion of an action like a DMA transfer, packet transfer, or similar operation with an interrupt towards the CPU. In general, the timing of such an interrupt should be similar to what one would see on real hardware. Note that some driver software crashes if the response comes immediately, as it is not built to handle an interrupt in the middle of the send routine -- it assumes a certain delay before a completion interrupt and does not protect itself against it.

7.2 Cluster Events

If the device performs a series of small related events, it is desirable to cluster these events into larger chunks, even if the simulator cannot respond immediately. For example, in a DMA transfer, rather than moving a few bytes every single cycle, the simulated device can move a whole memory page at a time every N cycles, where N is adapted to give the same overall transfer rate. Again, this means the model is invoked fewer times, and furthermore it will trigger other devices less often, too.

7.3 Transform Events into Functions of Time

Continuous events or events that occur regularly should be avoided. Instead, their effects should be computed (based on the elapsed CPU time) when they become visible. The archetypal example of this is a cycle counter: instead of invoking the model to update the value on every cycle, the model should compute the value on demand whenever the counter is read. If the value is read every N cycles, this replaces N increments with one subtraction. If the value is never read, no computation is performed at all.

This principle is valid even if the computation takes as much or more time than the sum of the individual updates: if the value is never needed, it will never be computed, and even if it is, it is usually more effective to optimize the model by reducing the number of invocations than by reducing the time spent in each invocation.

Chapter 8

Testing Device Models

How do you test a completed model to see if it is accurate and efficient enough? Basically, you try to find software that stresses the device as much as possible. Try different OS/driver combinations (if this is a requirement). Find programs on top of the OS that exercise the device in different ways. Perhaps there are diagnostic programs that verify that the device functions correctly. It is a good idea to run the programs on real hardware first, to make sure the software is functional, before trying them on the virtual hardware.

Run the selected programs in Simics, with the device model in place, and look for signs of trouble. Such signs may be

1. The device reports accesses to unimplemented features. These will need to be modeled.

2. The program, OS, or simulated machine behaves strangely, indicating flaws in the device model's functional behavior.

3. Simulation performance is poor. The model needs to be made more efficient.

It is also a good idea to sometimes let the tests run for a longer time. This allows you to spot problems with, e.g., memory leaks in the model implementation, or diagnose situations where the driver has entered some fall-back mode after seeing too many errors. The `VTMEM` package can help in diagnosing memory-related problems (see the *Simics Reference Manual* for details).

Chapter 9

General Usability

9.1 Device Model Style

The user interface of a Simics module consists of three parts: its attributes, its interfaces, and the commands it adds to the Simics command-line interface. You should try to make the user interface of your model similar in style to that of existing Simics models.

Every model should have an **info** command, giving static information about the device, and a **status** command, that gives dynamic information. See the *Simics Programming Guide* for more information.

Look at the interfaces of similar devices to see what other commands may be useful, and try to use the same names for similar commands and parameters. Use existing API functionality where appropriate, rather than inventing your own.

9.2 Checkpointing

The ability to checkpoint and restore the state of a system is crucial to Simics functionality. Your device model should support checkpointing. In particular, you should ensure that:

- All relevant state can be accessed as attributes
- The device can have its entire relevant state restored by writing to those attributes
- That an attribute can be written to any number of times while the device is active, and not assume that only a single initialization will be performed (to support the quick checkpoints of Simics Hindsight)

Attributes containing information which is set when a configuration is first loaded, have historically been coded as not being dynamically settable (as changing such information on the fly could compromise the stability and consistency of the simulation). Their contents are supposed to maintain the same value over the entire life of a Simics process; thus, updating them does not make sense. In order to support Simics Hindsight, however, it is necessary to make these attributes settable. The solution is to check that the value being set is the same as the previous value (i.e., the one set upon initialization), and if not, report an error.

Furthermore, device models often cache parts of their state internally for the sake of speed, rather than keeping it accessible via attributes. To ensure correct behavior, e.g. when

reversing to a micro-checkpoint, all internal caches have to be cleared when an attribute is set.

In the case that a module lacks any attributes that can be used to indicate flushing of the state, there is a special interface `temporal_state_interface_t` available that lets a module get callbacks when the state is being reset for Hindsight micro-checkpoints. This is however very rarely needed.

9.3 Error Reporting

The model should handle errors in a forgiving way. Avoid crashing or triggering assertions; instead, log an error message and try to continue anyway. If necessary, halt the simulator, but allow the user to continue manually.

There are three different kinds of errors that should be reported by a Simics device:

1. *Outside architecture* error. A part of the device whose behavior is not specified in the hardware documentation has been accessed. For example, a reserved register has been written to.

2. *Unimplemented* error. A part of the device which has been left unimplemented in the model (abstracted away) was accessed. This suggests a bug in the model, or that the model is used with software it was not developed for.

   In some cases it is sufficient to give a warning for this kind of situation, for example if the model returns approximate or invented values.

3. *Internal* error. The internal state of the model implementation has been corrupted beyond repair. Look for "this will never happen" comments and reconsider...

Simics has extensive support for logging, allowing you to assign the output to different message categories, and different levels of verbosity. See the *DML Reference Manual* and the *Simics Reference Manual* for details. Logging is mostly used during development and debugging of the model, but is also useful to aid inspection of the device state during actual simulation.

Always use detailed error messages. Often, the error messages are the only source of information when a bug is reported by another user. It is not certain that the execution can be repeated, or that the developer will have access to the same setup.

Index

Virtutech, Inc.

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

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

http://www.virtutech.com