

**SIMICS**  
**MODEL BUILDER**

**SIMICS SYSTEM  
MODELING**

**MARCH 2009**

JAKOB ENGBLOM

[WWW.VIRTUTECH.COM](http://WWW.VIRTUTECH.COM)

## INTRODUCTION

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

The key to realizing the benefits of Virtualized Systems Development for a particular system or application is the ability to quickly develop high-performance virtual platforms. This is the task of *system modeling*, and this Virtutech white paper will discuss how it is supported in Virtutech Simics.

At core, Simics is an extremely fast transaction-level model (TLM) simulator. Simics features an efficient simulation infrastructure that has been honed by active use for more than ten years, very fast processor simulators, optimized target memory handling, and a proven API for device modeling<sup>1</sup>. All Simics models are transaction-level, in all their interfaces.

## MODELING AN ELECTRONIC SYSTEM IN SIMICS

When a new system is being addressed for virtualization using Virtutech Simics, the process roughly follows the following outline:

1. Map the system, by collecting and reading design specifications, programmers' reference manuals, and other relevant documents. This creates a list of devices and processors that make up the system, and how they are connected.
2. Based on an analysis of the foreseen system usage, make a preliminary decision on the necessary level of modeling of each device. Can it be

---

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

ignored, stubbed out, or does it need to be fully implemented? If the software load already exists, analyze it to see what it actually uses. If the software is yet to arrive, the system deployment planning or marketing plan for a new SoC can be very helpful to determine initial priorities.

3. Reuse existing device models and processor models from the Virtutech library. The library makes it faster to produce an initial model, since models for many common standard parts already exist. Reuse often means adapting an existing model of a similar device, which is much faster than writing a new model from scratch. Virtutech also provides frameworks for writing models for devices attached to common system interconnects like PCI, PCIe, RapidIO, Ethernet, I<sup>2</sup>C, and others.
4. Create initial models using the Virtutech Device Modeling Language (DML)<sup>2</sup>. Ignore and simplify as much functionality as possible initially to quickly get to a basic model.
5. Test the new system model. If there is already software available, test using that software, and add any missing functionality or devices required by the software. For new devices where no software yet exists, create independent test cases that exercise the specification of the software-hardware interface from the software side. The cases to test are guided by the usage analysis from step 2.
6. Iterate until the model runs the available software or passes all defined test cases.
7. Evolve the virtual platform as the physical hardware design stabilizes and is updated during a design project or system life cycle.

This is a classic iterative software-development methodology, where you test the software being developed – the hardware model – early and often in order to explore the precise requirements. It goes by many names, from spiral model to agile methods and test-driven development. The iterative approach can be applied to help develop and debug the hardware design, considering the

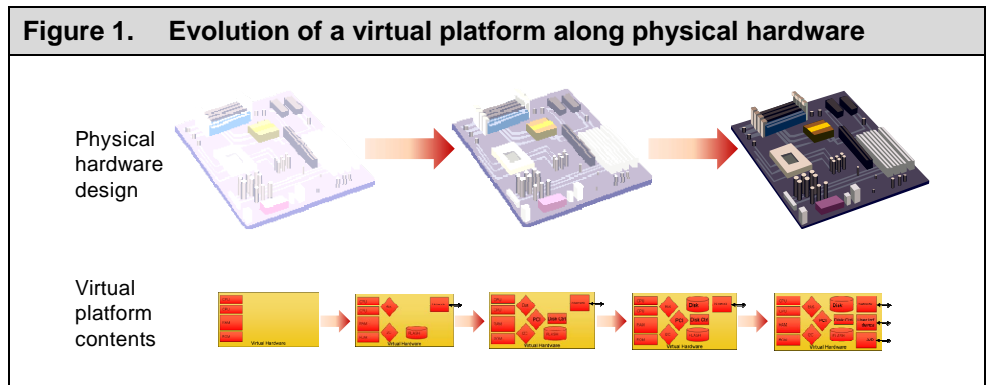
---

<sup>2</sup> For more on Virtutech DML, see the [Virtutech white paper on DML](#).

hardware model as a software object that can be updated to reflect new and better understanding of the target domain and software interface requirements.

Virtual platform design can be and is usually started before the final hardware specification is frozen. This is not a problem, as the virtual platform can be updated as the hardware specification changes or is completed.

As shown in Figure 1, as the physical hardware design evolves, changes, and matures towards completion, the virtual platform adapts and aggregates more content. Thus, the virtual platform can be used as an executable specification of the hardware, a specification that can run software and be used to validate and evaluate the design.



During the hardware development process, the virtual platform is used to start software development early, and also to get feedback from the software team to the hardware developers. This is an important added value from building a virtual platform for a new piece of hardware that should not be overlooked.

Often, it is possible to start using a virtual system almost immediately after starting to develop the model. Even a basic system that does not yet contain all components can be used to get development started. For example, a boot loader typically requires less virtual hardware to be in place than a full operating system port. Such a limited system can be used both as an early start for a design that is finalized, or as the early approximation for a system design that is not yet final.

Each individual device model can also start out quite simple, implementing only the basic operation mode(s), and later have more complex optimized

operating modes added. Over time, more devices and more details for each model will be added to the virtual system, evolving towards the final model.

It is always possible to go back and improve the model thanks to the clear modularity of the Simics system, and its robustness for partial models.

### ***Modeling the Hardware-Software Interface***

The Simics approach to system modeling is to focus on the *hardware-software interface* and model whatever is needed to make the software side perceive the virtual hardware as just another piece of hardware.

Modeling a system at the hardware-software boundary has several practical advantages:

- This level is often the best documented and most stable layer in the system. Since hardware design and software design are usually in different teams, documentation is necessary. It is an exposed interface in the physical machine, and therefore considered external by the hardware group.
- It is the natural integration point between hardware and software engineering teams, and between companies producing hardware and their customers building software.
- Device models can often be constructed with only the details provided by a *Programmer's Reference Manual, PRM*, since it specifies what the hardware should do from the software perspective. Detailed documentation on the actual implementation is typically not needed. You do not want or need documentation at the bus signaling level, as that is too low-level for a high-performance virtual platform.
- Modeling times are minimized as only the functionality actually used by the software needs to be implemented.
- Low-level hardware implementation details, such as those defined by RTL, are not required making modeling much simpler and faster.
- Simulation performance is optimal because the simulation functionality is implemented with the minimal amount of state change needed, and with minimal timing details.

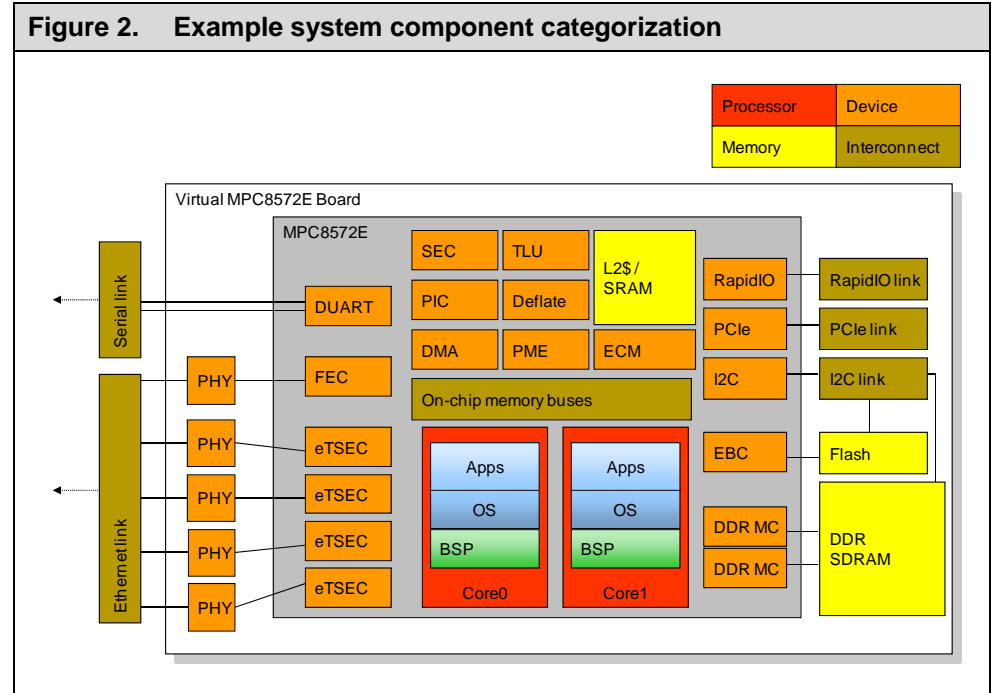
- System modeling can start once a system design is semi-complete, and proceed in parallel with the hardware implementation effort. This helps software developers get access to virtual hardware long before physical hardware is ready for use.

### ***Anatomy of a System Model***

To understand how to model a system in Simics, it helps to break down the contents of a hardware system into four categories of components:

- Processor cores – the CPUs actually running processors instructions. For example PPC 464, MPC e500, Core 2 Duo, MIPS 5Kc, or ARM9.
- Interconnects – networks and buses connecting devices, machines, boards, and cabinets together. For example memory buses, peripheral buses, serial, Ethernet, I<sup>2</sup>C, PCI, SCSI, USB, or MIL-STD-1553.
- Memory – RAM, ROM, EEPROM, FLASH, and other types of memory devices that store large amounts of code or data.
- Devices – anything else. All the peripheral units that move data between machines or do work that is not instruction processing. Timers, interrupt controllers, ADC, DAC, network interfaces, I<sup>2</sup>C controllers, serial ports, LED drivers, displays, media accelerators, pattern matches, table lookup engines, and memory controllers are just some examples of devices. Often, devices form part of an SoC along with processors, memories, and interconnects.

Figure 2 shows an example categorization of the components found in a Freescale MPC8572E SoC design, along with some parts of a virtual board incorporating it. Note how the number of devices is much higher than the other types of components.



In general, most of the work in creating a new system model is spent modeling devices, and they are the most numerous and least standardized of the hardware components. Most systems in Simics contain a large set of devices reused from existing model libraries, but there are almost always some new devices in any custom system that has not yet been modeled.

In contrast to devices, processors tend to have fairly few variants. They are expensive to model as they have large instruction sets and are much more complex than all but the most extreme devices. Creating a truly high-performance processor is also hard work. Thus, most Simics users use the processor models provided by Virtutech to leverage the existing investment in high-performance, well-tested processor models. It is also possible to create you own processor models and plug them into Simics (or reuse existing processor models that already exist in an organization).

Memory systems are another special case. They need to be tightly integrated with the processor models in order to provide maximum performance, and are very generic in nature. The representation of memory in the simulation also need to support demand-based paging, 64-bit addressing, zero-page detection, change tracking (for incremental checkpointing) and other fairly complicated

features to make it possible to model really big real-world systems. Simics users normally use the memory implementation provided by Virtutech, since this is robust, proven, high-performance, and supports checkpointing and scalability features.

Interconnects are also comparatively few in variants and standardized. Thus, they are normally easy to reuse across target systems. For Simics, they are usually provided by Virtutech since once again, a fully-featured implementation is fairly complex. Interconnect models have to support distributed and multithreaded simulation<sup>3</sup>, as well as connections between virtual and physical networks and support for traffic record and replay for reverse execution.

In addition, Simics virtual hardware setups tend to contain features such as debugger connections, instrumentation modules, and fault injection. These are not really part of building a system model, however, so we will not discuss them further in this paper.

### ***Anatomy of a Device Model***

As stated, most of the work involved in modeling a hardware system is invested in creating the device models unique to that system. To allow software to run completely unmodified, the model has to mimic the properties and behavior of the hardware, at the point where the software interfaces to the hardware.

This is the hardware-software boundary, and the devices in the system typically show up as banks of control registers at certain locations in some processor memory map. Device drivers in the software read and write these addresses. This is the *memory-mapped programming registers* shown in Figure 3.

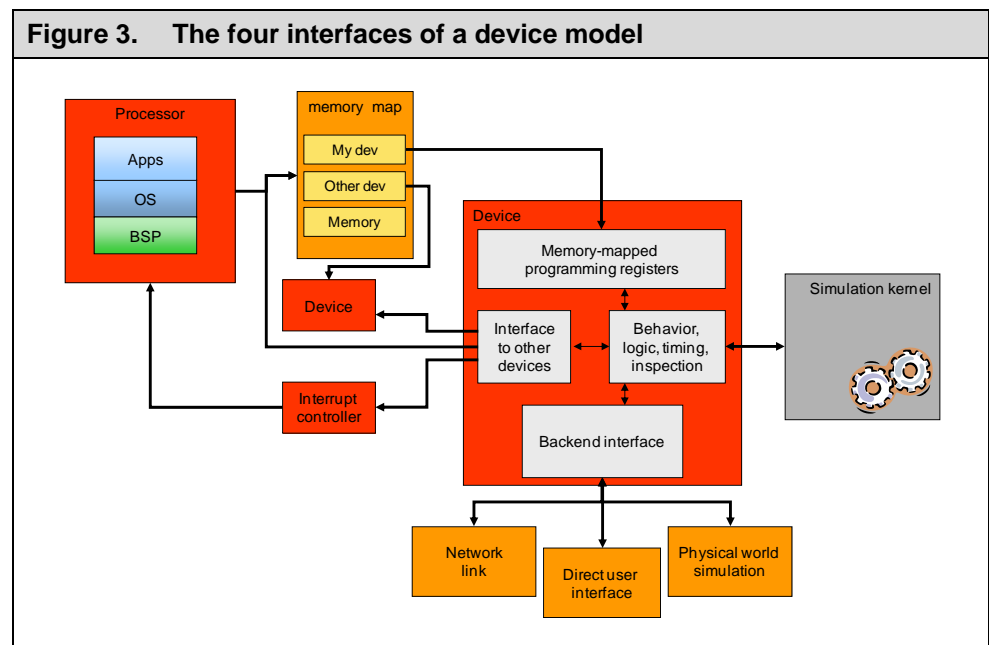
A register map can be very simple like the eight one-byte registers used to control an NS16550 serial port. It can be spread out in several parts across the memory map of the processor like a PCI device which maps both a configuration space and a memory space. The register map can be very large,

---

<sup>3</sup> For more on threaded and distributed simulation in Simics, see the [Virtutech white paper on Simics Accelerator](#).

like that of a Freescale MPC 8260 CPM which has thousands of registers. Simics and the DML programming language make specifying and implementing even large, complex register mappings easy and fast. Sometimes, devices read description tables based in memory, as a way to handle complex programming tasks that are hard to express as sequential accesses to programming register maps.

Behind the programming register map, the *behavior* of a device model has to be defined. At a minimum, this needs to have enough functionality that the software will work. Simics device models are passive software objects that only act when something happens, so the behavior is coded in a reactive, event-driven way. Most of the work is performed in callbacks. The interaction with the simulation kernel mostly takes place from the behavioral part of a model, as shown in Figure 3.



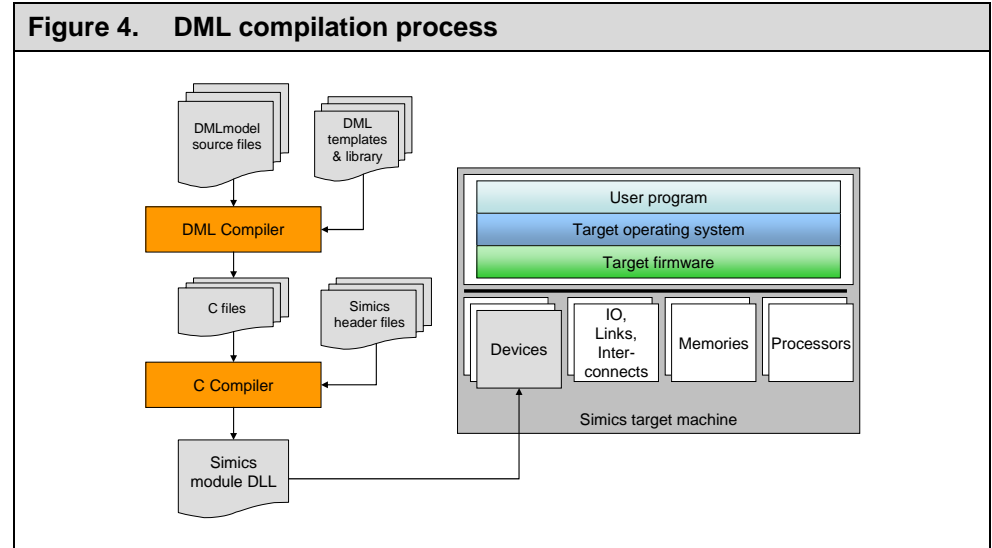
In most cases, there will also be a *back-end* part of device, where interactions with the environment are handled. For example, a model of the buttons on a watch has to provide some means to “press” the buttons. A screen device needs to display what is being drawn on the host machine screen, and a network device needs to send and receive packets. A sensor needs to pick up values from a simulated world to report readings.

In Simics, this is normally implemented by a transaction-level interface (function calls) to some network link object or user interface object. The norm is to keep the direct user interaction encapsulated in its own object or set of objects, so that device models are independent of the details of the user interface implementation. This indirection makes it easy to record and replay user actions, supporting traffic replay and reverse execution. Finally, it is key to enable multithreading of a simulation, as the network link objects encapsulate all data exchange between the parts of a simulation that are running in parallel.

Finally, a device model often interacts with *other devices* in the system. There can be DMA accesses to read and write memory, interrupts raised towards a processor, or direct data transfers to tightly-coupled devices. For example, a multiprocessor system controller will need to route interrupts from devices to processors and pass interrupts between processors in the system. Network processing accelerators have direct data connections to the network interfaces of an SoC, where network frames go directly without touching system memory. All such interfaces are modeled using transactions in Simics, and effected as events, not continuous processes.

### ***Device Modeling Language – DML***

An important enabler for the Simics methodology and speed of model development is the *Device Modeling Language, DML*. DML is a declarative language specifically designed to speed TLM device modeling. It provides a concise syntax and language constructs specifically targeting the device modeling problem at the TLM level.



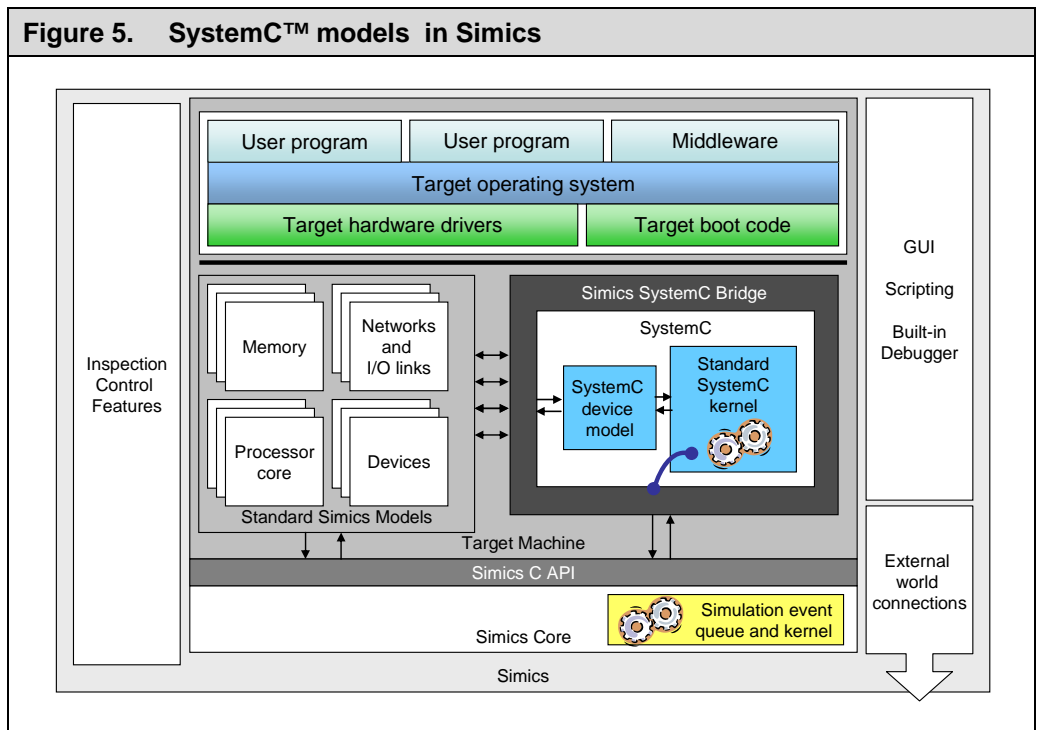
The *DML Compiler* generates a C code implementation of the device modeling calling the Simics API and automatically supporting Simics features like checkpointing, logging, and reverse execution. This process is transparent to DML users. The compilation flow is shown in Figure 2.

DML models are much more concise than corresponding C language models, and much easier to write, read, understand, and maintain. DML also has explicit support for an iterative development of device models by providing default implementations for many device register behaviors and marking unimplemented registers with explicit flags. DML also provides for documentation of a model inline in the source code, and the documentation is extracted and used to help Simics provide a better end-user interface for a model.

### ***Model Implementation Languages***

DML is the recommended programming environment for new device models in Simics, but any language that can call the Simics API can be used. Simics comes with support for C, C++, and Python. Python can be used for prototyping and higher-level functionality that does not require high performance. C and C++ are used for code that needs to implement very complex algorithms where DML does not really provide any expressive power above that of C and C++. C and C++ code is also used for integration of existing models into Simics. Existing transaction-level models written in C

can be used as the basis of Simics models, saving modeling time and reusing existing models.



SystemC™ models can also be used in Simics, thanks to a Simics to SystemC Bridge. As illustrated in Figure 5, SystemC is integrated into Simics by including a standard OSCI SystemC 2.2.0 kernel as a subsystem along with the SystemC device models that will be used. There is no need to modify the SystemC device models to work inside Simics, even if that is possible to make use of particular Simics features from within SystemC. The SystemC Bridge takes care of coordinating and synchronizing the activities of the SystemC kernel and the Simics kernel, and moving transactions between the SystemC subsystem and the rest of the Simics system.

For best performance, the SystemC models should be transaction-level models coded using a framework like TLM-2.0, and at a high level of abstraction as discussed in the next section.

## SIMICS TLM ABSTRACTION LEVEL

The functionality of a device model in Simics is implemented using *transaction-level modeling*, *TLM*. In TLM, each interaction with a device (for example, a write to or read from the interface registers of the devices by a processor) is a single simulation step: the device is presented with a request, computes the reply, and returns it in a single function call. This is far more efficient and easier to program than modeling the details of how bits and bytes are moved across interconnects, cycle-by-cycle.

In general, immediate completion of an operation is sufficient for modeling device behavior. However, if the software using the device expects a delay, that delay must be modeled. A classic example is that of a device driver that writes a device control register that initiates a process that will later lead to an interrupt being triggered. Commonly, device drivers are not coded defensively enough and expect that there is time to execute some more code before the interrupt triggers. If the interrupt triggers immediately, the driver breaks. This is strictly speaking a bug in the device model, but it works on current hardware due to the large-scale timing of that hardware. Thus the device model must mimic this behavior and wait a short while before generating the interrupt.

In Simics, such deferred actions are implemented by posting events to the Simics kernel, and the device later gets a callback (from the kernel) when it is time to effect some state change to reflect the event. A typical example is raising an interrupt towards a processor, or moving data between two simulation devices.

Common names in industry and academic discourse for the Simics TLM style is *PV*, *Programmer's View*, or *LT*, *Loosely Timed*, TLM. Compared to the OSCI SystemC TLM-2.0 coding styles and abstraction levels, the Simics default level of abstraction corresponds to a streamlined form of LT where device models never consume time inside a single transaction from the processor and never call the SystemC kernel *wait()* function. Timing annotations on memory accesses are typically assumed to be zero in Simics.

### ***Memory System Modeling and Address Maps***

Simics features a very efficient way to model memory systems and the bus interface of the processor. The address space of a processor is modeled as a *memory map*, where memory reads and writes are directly routed to the

### Who Should Do the Modeling?

Virtutech is often asked about the skill set needed to create these models. Is it most appropriate to task a hardware engineer or a software programmer with model creation? We usually suggest that the customer look at the group of individuals responsible for writing the firmware, device drivers, and doing hardware bring-up. This is the group usually most familiar with the Programmer's Reference Manuals for the various devices of their system, and they understand how the software interacts with the hardware. These individuals make excellent candidates for doing device modeling!

If hardware engineers are tasked with the work, then it is important for them to be strong programmers because creating models is fundamentally about writing software. Furthermore, it is important to be aware that Simics models should not reflect the implementation details of the hardware, only its functionality. This might require a paradigm shift for a hardware engineer.

In general, the following skill set is needed for simulation engineers:

*The ability to understand a Programmer's Reference Manual.*

*An understanding of how the hardware has been designed.*

*An understanding of how hardware and software interact.*

*Proficiency in C or a similar language.*

recipient memories and devices without involving any explicit models of buses and bridges between buses. The memory map is a fundamental service provided by the Simics framework, and core component in enabling very fast simulation performance.

Simics has highly optimized the way processors access memory, in order to enable very fast instruction execution and data access. In a virtual system, using RAM and reading from FLASH and ROM is very fast and does not involve any explicit device models. The memory handling is part of the Simics core, and is exposed to processor models using the processor API.

A memory controller model in Simics exists in parallel with the memory itself. The memory controller model will implement functionality such as changing how memory is mapped by changing the memory map setup in the simulator, but it is not involved directly in regular memory accesses. Simics memory maps are always 64-bit capable.

Simics also provides very capable *memory images* to handle the content of memories. These images provide lazy allocation, so that only memory which is actually used is represented. This allows Simics to simulate memories which are much larger than the physical memory of the host machine. It also automatically supports incremental checkpoints of the system state. Images also provide the ability to share identical pages between different images, reducing the memory consumption of the simulator. The same image system is used for RAM, FLASH, disks, and any other larger digital storage. Just like memory maps, Simics memory images are always 64-bit addressing capable regardless of the host architecture. This simplifies the coding of 64-bit models even when they run on 32-bit hosts.

For PCI and similar interfaces where several levels of addressing are being used, Simics uses subordinate memory maps cascaded from the primary memory map. This makes it easy to translate real-system mappings into the Simics system configuration. PCI models in Simics support software probing and configuration, just like real PCI systems. The software setup is then reflected in the makeup of the PCI memory map, and device accesses are still very fast.

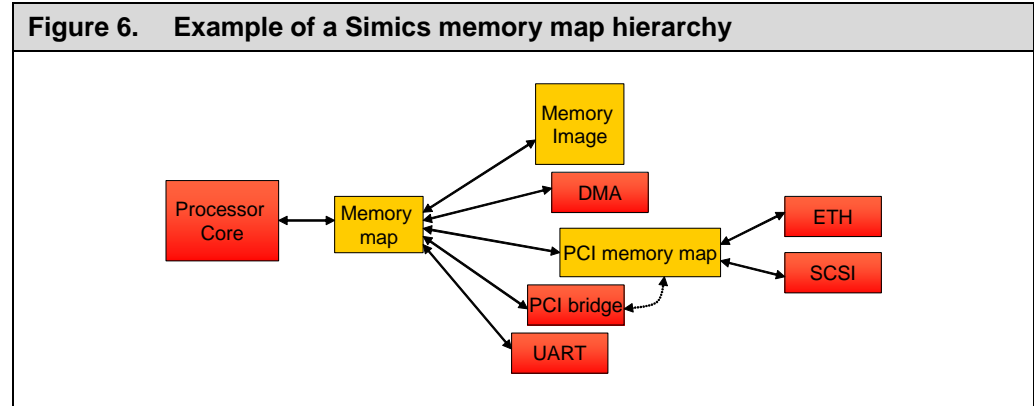


Figure 6 shows a simple example of how Simics uses memory spaces to map the processor’s memory range to its immediately connected memory and devices, as well as across a PCI system to Ethernet and SCSI controllers. The PCI bridge will configure the PCI memory map when the software configures the PCI system, but will not be invoked in actual accesses to PCI devices. Once the PCI mapping is setup, the Ethernet and SCSI controllers are as quickly as any other device from the processor.

### ***Processor Modeling***

In a simulation at the Simics level of abstraction, a large majority of the simulation execution time will be spent in the processor models running code. Processor models are highly optimized with tight integration with the simulator memory system in order to reach peak speeds of many billions of simulated instructions per second.

In order to run all the software of a system, the processor must implement both user-level and supervisor-level programming models. The memory-management unit is also needed, along with other model-specific registers and other low-level interfaces visible to the software in some execution mode. Fundamentally, anything readable or writable from software has to be modeled.

Simics system models and processor models do not usually attempt to model the precise cycle timing of code execution as dictated by the processor pipeline, cache hierarchies, and memory bus contention. Virtutech-provided processor models allow the speed of a processor model to be set by the processor clock frequency and the cycles-per-instruction (CPI) property of the

processor. Typically, setting CPI to one, i.e. executing one instruction every cycle, is a sufficient solution. Setting CPI to other values like 2 or 3/2 can help more closely approximate the average instruction execution rate of a certain processor for the software in a certain system. Experience shows that this works well for almost all workloads in almost all circumstances.

Processor models are created and supplied by Virtutech, or by third parties using the processor API that has been available since Simics 4.0. There is no speed penalty for processors using the processor API, Virtutech models use the same API for connecting to the simulator core. Virtutech provides a large library of fast and functionally complete and correct processor models for most common embedded and desktop architectures including Power Architecture, MIPS, ARM, SPARC, and x86, in both 32-bit and 64-bit variants. Please see the Virtutech web page for a complete and up-to-date list of processors supported at any particular point in time.

The DML language is not suitable for modeling processors, it is tailored for modeling the devices that the processor (or processors) in a system is connected to.

### ***More Detailed Models***

In addition to the default fast Simics model of simulation, Simics can support simulating a system at a greater level of timing detail.

- Simics can be run in a slower simulation mode where all memory accesses can be annotated with delay time in order to account for memory access delays and the effects of cache systems.
- Detailed in-house models of processors can be used in a Simics system context, using the processor API.
- With Simics in hybrid mode, some or all parts of a system can be modeled at a clock-cycle level of detail.

Simics can use fast functional models to position a workload and then change to more detailed models for in-depth studies. In this way, much larger workloads can be efficiently handled than is possible with only detailed models. When studying parts of a system at a high level of detail, other parts can be left at a functional level. This provides a faster implementation route to a model complete enough to run a real software load, and also makes the execution itself faster.

## TARGET SYSTEM STRUCTURE

We now know how to model individual devices, processors, memories, and interconnects in a system, and how to connect them using memory maps and interfaces. It is possible to build the complete system model from these pieces directly, but in most cases more structure is needed to help in system configuration, inspections, long-term evolution and reuse of subsystems.

To support this, Simics features a hierarchical *component system*, where components are an added abstraction corresponding to the physical or logical grouping of devices into chips, boards, racks, networks, and systems of systems. Typically, a component corresponds to a physical unit such as a board, SoC chip, memory module, hard disk, or other unit that is typically found on a system block diagram. They can also correspond to useful logical groupings of devices that recur in multiple systems.

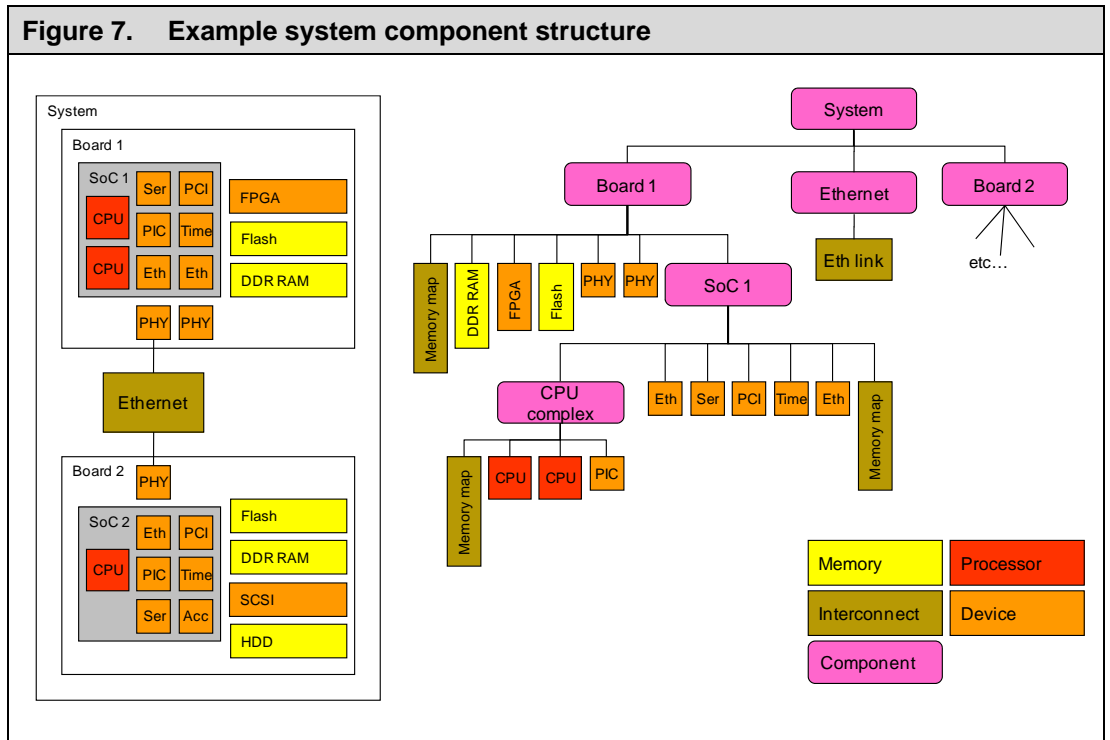


Figure 7 shows a somewhat abstracted example of a system decomposition into components. The system consists of two boards connected by an Ethernet

network. They each have a different type of SoC on board, along with memory, network PHY connectors, and an FPGA or a SCSI disk controller. The left-hand side shows the physical structure that we want to model and the right-hand side the corresponding Simics component model.

The fundamental simulation units of processors, devices, memories, and interconnects that we discussed above are structured into components corresponding mostly to the physical packaging. However, since we have realized that the CPU complex in SoC1 tends to recur across different related chips, it has been given its own component, to make it easier to reuse.

Simics components encapsulate system configuration issues such as how to connect devices to each other, to the memory map, and to the interrupt system. In the example in Figure 7, note how Simics memory maps are present at all levels of the component hierarchy, since each component tends to provide more devices that need to be mapped into memory, and that device models can appear as leaf nodes in the tree at all levels.

Components furthermore expose easy-to-use system-level parameters like memory sizes, processor speeds, and number of processors. They offer a way to check constraints on values, and to hide just exactly how each parameter is implemented. For example, configuring a DDR3 memory module for a certain size requires the component to compute the bank counts, rank sizes, and other parameters that are communicated via I<sup>2</sup>C to the memory controller. But the external configuration interface is just the size of the memory.

The Simics component system allows the drawing-out of connections from subcomponents to be exposed at the top level. For example, the PHYs

## PREPARING FOR SYSTEMA MODELING

Before building a system model, certain information should be collected. Since obtaining documentation can sometimes take time, it is a good idea to start looking for documentation and information as early as possible in a modeling project. Documents and information that Virtutech has found to be particularly useful are:

- System documentation, including block diagrams, describing how the components of the system are connected.

- Programmers Reference Manuals, PRMs, for each of the main chips, devices, or functional blocks. The PRM goes by many names, some others being *Programming Reference*, *User Reference Manual*, *Technical Reference Manual*, *User Manual*, and *Reference Manual*.
- Low-level source code (BSP, device drivers, firmware, operating systems) of the software that manipulates the hardware, if it already exists.

Once a basic understanding of the system has been obtained, you need to prioritize the components that need to be modeled. There are some things that need to be considered initially:

- Try to identify the minimum set of devices and processors required to boot the software. Having built such a base, it is then easy to incrementally add devices to the system. Note that the software sometimes has to be reconfigured to use only part of a system, and that this is a perfectly reasonable temporary measure.
- Try to identify the simulation components where the most technical risk is found: risks in developing the model, as well as in the software that manipulates the component.
- Try to identify the components that have no initial impact on your system, i.e. those that will not be manipulated or used by software.
- Try to identify the memories in the systems that can be handled using Simics default memory modeling.

These tasks are really an application of the modeling philosophy and principles described in below.

## SYSTEM MODELING GUIDELINES

Over the years, Virtutech has obtained significant experience in how to build virtual systems. This experience can be distilled into a small set of fundamental design principles:

- Follow the hardware

- Follow the software
- Follow the boot order
- Don't model unnecessary detail
- Reuse and adapt existing components
- Develop device models in DML

### ***Follow the Hardware***

The main design principle used when creating Simics models for a system is to follow the structure of the physical hardware closely, while abstracting functionality where possible. The overriding goal is to ensure that the software developed on the simulated system will run on the physical hardware and vice versa. This includes variations of the hardware configuration. Thus, all software-visible functions have to be accurately represented in the simulation, and the easiest way to ensure this is to design the simulation model using the same component and communications structure as the physical hardware.

The components to use as the basis for understanding and decomposing the hardware system design are typically entire chips for a board design, and function blocks inside a System-on-a-Chip (SoC). From a Simics perspective, an SoC or a board are really equivalent – they both group a number of devices together with a set of processors, and provide interconnects between them. From a practical perspective, a board often contains some SoC devices, and this leads to a recursive process where the board is broken down into SoCs, and the SoCs into devices.

The starting point is thus the layout of a board and the block diagram for an SoC, as presented by the programmer's reference manuals (PRM). An important source is the memory map typically found in the PRM for both boards and SoCs, showing how devices are presented to the core processor(s) in the system.

Note that some components might be addressed indirectly and not have their own slot in the memory map. A common example is serial EEPROMs accessed over I<sup>2</sup>C from an I<sup>2</sup>C controller. The EEPROM is not visible in the memory map, but it is still accessible to the processor and needs to be considered in the model.

The ultimate goal is to have a list of the *devices* that make up a system and a map on how they are interconnected.

The interconnections between devices also need to be considered in order to have a good breakdown of a system for modeling. Some interconnections are usually invisible to the software, and thus do not need to be modeled in Simics. A good example are on-chip device interconnects like AMBA, CoreConnect, and OcEAN used in various SoC designs. These interconnects are implemented using complex crossbar technology and bus arbitration which is not visible at the Simics default level of modeling. Also, the hierarchy of buses used in interconnects like AMBA with its high-speed and low-speed buses is invisible. Simics goes straight to the resulting memory map. As far as Simics is concerned, all interconnects just transport bytes from one place to another, and that is modeled by mapping devices into the memory map of a processor.

Interconnects that go outside the memory map do need to be modeled explicitly, however. Typical examples are I<sup>2</sup>C and Ethernet networks, where it makes sense to model the network moving addressed packets around as an entity in its own right. PCI is another case, which was discussed above.

### ***Follow the Software***

Implementing every available function of a system in order to be complete in following the hardware is usually not necessary in order to run a particular software load. Instead, we should only implement the functions actually used by the software which we want to run. This is commonly the case with integrated chips and SoC devices which contain more functions than are used in any particular case.

When implementing models of new hardware, you should try to target some particular use case initially, and then broadening to other use cases. From a project planning perspective, this means deciding on a pilot software group which is the first to receive the virtual model and work to fulfill their requirements first.

In a concrete application of this principle, the Freescale MPC8548 SoC was used as a controller chip for a custom ASIC on a custom board. The MPC8548 has a rich set of external connections such as PCI express, RapidIO, Ethernet, I2C, MDIO, and others. In this particular case, the RapidIO functionality of the 8548 was not used, and thus that function could

be left out from the initial modeling effort for the MPC8548. When other systems appeared that used RapidIO, the function was added.

Another example is the Marvell MV64360 integrated system controller. This controller contains a memory controller for a PowerPC processor, along with some generally useful functions like PCI, serial ports, and Ethernet. Many boards using this controller do not use the built-in Ethernet port, but instead they use an external Ethernet chip connected over PCI. In such a case, the built-in Ethernet controller does not need to be included in the model of the board.

Sometimes, the software explicitly turns off some functions or devices on a chip, and in such cases one or more control registers have to be implemented that accept the “off” setting, and give a warning if any other status is written.

It is also good practice to include registers corresponding to unimplemented functionality in the model. Such registers should simply log a warning when they are accessed. Compared to leaving them out altogether, this explicitly documents design assumptions and modeling decisions in the source code. It also provides an obvious indication when the assumptions are violated.

Within a device, only the *functionality which is actually used by the software* should be modeled. This typically means focusing on a few operation modes or functions of a device, and leaving the rest unimplemented (but explicitly so, as discussed below). Often, modeling starts with a completely unimplemented device, looking at how the software interacts with the device to determine what actually needs to be implemented.

For example, a PCI Express bridge like the PEX PLX 8524 can operate ports in both transparent and non-transparent mode. However, if non-transparent mode is not actually used in a system, it can be left unimplemented.

Note that an effect of this style of modeling is that even though a device model exists, it might not fulfill the requirements of use in a different system from the one which it was developed for. As time goes on, a device typically gains functionality as it is subject to different uses from different target system configurations and software stacks.

### ***Follow the boot order***

In large and complex target systems containing multiple boards (or subsystems), it is necessary to decide the implementation order of the boards. To get to a working model that can run some meaningful software as quickly as possible, the best strategy is to follow the boot order of the system. This means that the first board or subsystem to model is the one that takes charge upon power-on of the system, and that other system components are modeled later.

This is especially relevant when modeling new hardware, since the software groups will want to start working with the system boot first. That part is the most hardware-dependent and thus the one for which access to virtual hardware is the most pressing.

For example, in rack-based embedded systems there is usually some controller board or master board that boots first, and then provides software for data plane boards and slave boards in the rack. This controller board has to be modeled first, as the other boards are not of much use without it. The same pattern occurs in other places, for example DSP boards where a controller processor sets up software for the DSPs before booting them. Here too, modeling the component that boots first as the first step is the sensible choice.

### ***Don't model unnecessary detail***

It is easy to fall into the trap of modeling detailed aspects of the hardware that are invisible to the software. The execution overhead of modeling in too much detail can significantly slow the simulation. A simple example is a counter that counts down on each clock cycle and interrupts when it gets to zero. This should not be modeled by calling the timer model each clock cycle. Note that the counter is only visible to the software when it is explicitly read. A better implementation is thus for the model to register a call-back with the simulation kernel at its expiration time. If the software reads the register before this point, the model has to work out what would be in the register at that point by looking at the current simulation time. Such a model runs much faster than the detailed counter and is indistinguishable from it as far as the software is concerned.

A good Simics model implements the *what* and not the *how* of device functionality. The goal is to match the specification of the functionality of a device, and not the precise implementation details of the hardware. A good

example of abstraction is offered by network devices. In the physical world, an Ethernet device has to clock out the bits of a packet one at a time onto the physical medium using a 5/4 encoding process. In Simics, this can be abstracted to delivering the entire packet as a unit to the network link simulation, greatly simplifying the implementation. As far as the software is concerned, this makes no difference.

DMA controllers are another example of abstraction. In Simics, DMA is typically modeled by moving the entire block of memory concerned at once, and delaying notification to the processor (or other requesting device) until the time when the full transfer would have completed on the physical hardware. The bus contention between the processor and the DMA controller is not modeled, since this is not visible to the software. For a system architect with bandwidth concerns, a more detailed model can be created that logs the amount of data pushed, allowing bandwidth usage to be computed.

Abstraction can also manifest itself by making entire devices into dummy devices. For example, a memory controller might have a large number of configuration registers and reporting registers for factors like DDR latencies, number of banks open, timing adjustments, and similar low-level issues. The effects of these are not visible in Simics, and thus they can be modeled as a set of dummy registers that report sensible values to the software but writes to which have no effect on the simulation.

A nice side-effect of Simics-style modeling is that it is easy to reuse device models across multiple implementations of the device in question. As an example, the standard PC architecture contains a cascaded i8259 interrupt controller. Over time, the hardware implementation of the i8259 has changed from being a separate chip to becoming a small part of modern south bridges like the Intel 6300ESB. Despite this huge change in implementation, the same Simics model can be used, since the functionality exposed to the software is the same.

Sometimes, it is realized over time that some more details have to be added to certain device models. For example, some Ethernet network device models did not implement CRC error detection, but assumed that all packets delivered were correct. When the time came to model a system where the handling of erroneous network packets was a key concern, this was obviously not sufficient. Thus, the handling of CRC computation and flagging CRC errors had to be added to the models.

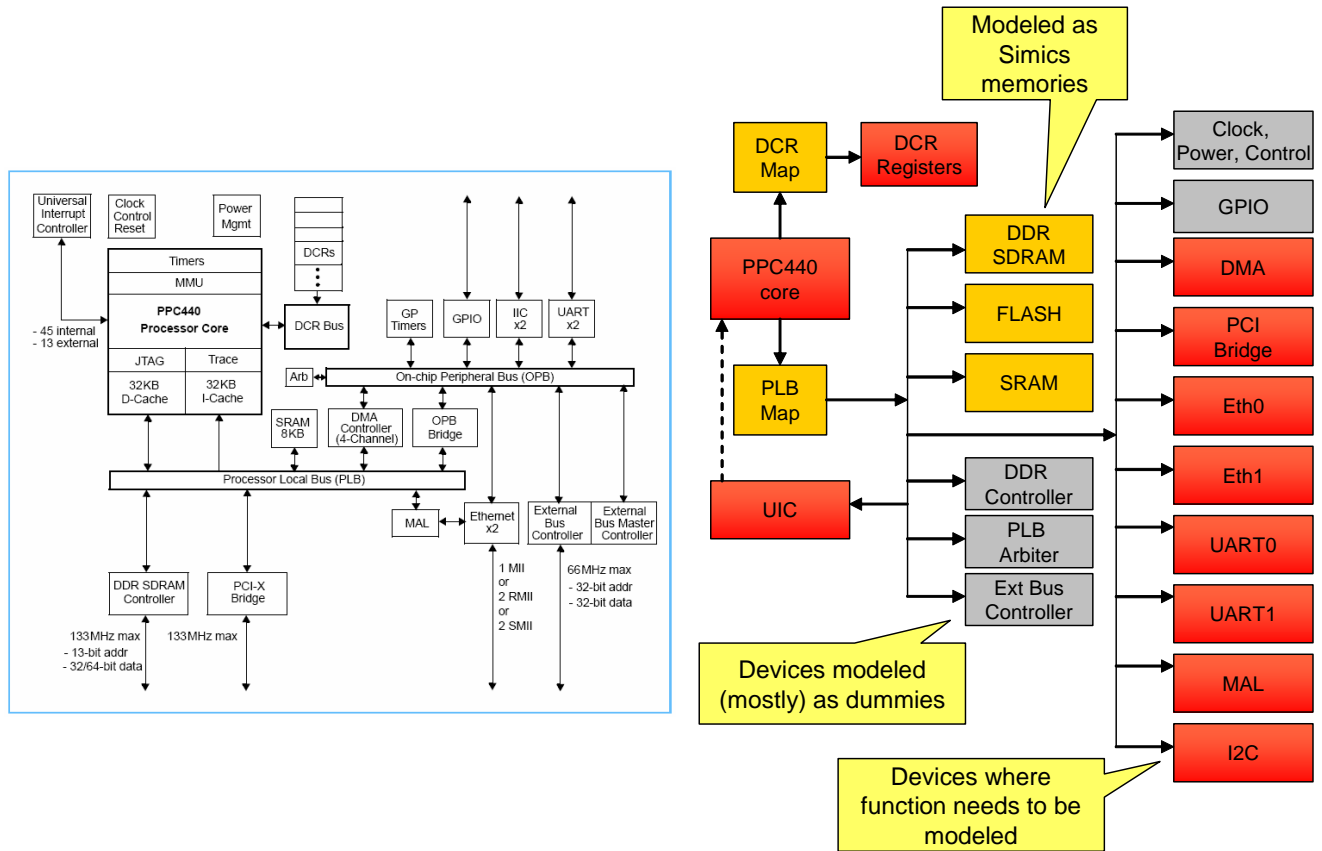
### ***Reuse and Adapt Existing Components***

Once a system has been analyzed and its devices and interconnections listed, it is time to start implementing all the required devices. At this point, reusing existing device models is very important to shorten the modeling time. Virtutech has a large library of device models which can be used to quickly fill in large parts of the functionality of a system.

Sometimes, the precise devices are not available in the library, but there are similar devices. In this case, the existing device can be used as a starting point and adapted and extended to create a model of a new device. Such adaptations of existing devices typically follow the path of hardware designers as they design successive generations of products.

The IBM Ethernet controllers found on the PPC 440GP and PPC 440GX SoCs, and also being sold as BlueLibrary IP blocks is one example of how one model has been reused and adapted to successive generations of hardware. Another example is found in Intel chipsets for Pentium processors; successive product generation share a significant amount of device functionality, even if the names of the chips change and the functionality is moved around between different chips.

**Figure 8. System mapping example for the AMCC PPC440GP**



Starting with the block diagram of the PowerPC 440GP SoC shown on the left and some knowledge of how the system is going to be used, the initial system map on the right was generated. This example illustrates several important points about how systems are modeled in Simics and the initial work to map a system for modeling.

The memory and bus controllers are modeled mainly as dummies, since they are not on the path to the memory image. The DCR register set is modeled as a separate memory map from the main "PLB" memory map. Some devices will have to be modeled to some extent to create a useful system model. The interrupt controller, UIC, is shown with arrow back to the processor core since it is allowed to interrupt it. Interrupts originate from the other devices in the system, but their connections to the UIC are not shown for the sake of simplicity.

Typically, adapting a device model involves either adding or removing registers, depending on whether we are moving to a less capable or more capable device. It is also commonly the case that some details in the memory map of the device change. Thus, the work of adapting a device starts with

comparing the programmer's manuals for the old and new device, and determining the following:

- Identical registers—for an adaptation to be worthwhile, most registers should fall in this category.
- Superfluous registers—functions in the existing model which are not found in the new device. These have to be deleted.
- Missing registers—have to be added to the new device model.
- Different registers—registers with the same function or name, but where the bit layout is different between the old and new device.
- Differences in register layout—the offsets at which various registers are mapped in the device memory map are different.
- Differences in the number of functions—some devices contain repeats of the same functionality, and the difference between devices is the number of functions implemented. For example, a different number of DMA channels or Ethernet controller ports. In this case, simply changing a parameter in the device model may be sufficient.

If there are too many differences, it may be more expedient and safer to implement the new device from scratch. As in all software development, doing too many changes to a DML model might be more work to get right than to implement the complete functionality from scratch (maybe borrowing some key source code from the old device model).

### ***Develop Device Models Using DML***

Finally, once we have reused and adapted existing devices and ignored any devices not used in a particular system, we need to create device models for the remaining devices. The document that describes how to program a device is often called the *Programmer's Reference Manual*, *PRM*, and the basic methodology of writing DML models is that of implementing the PRM.

As previously mentioned, the primary interface between software and the devices is the device registers. The PRM defines one or more register banks that contain the registers laid out at specified offsets. The register banks function as an address space for registers, such that one four-byte register may

occupy the address locations 0–3, another four-byte register occupies the address locations 4–7, and so on.

The method that Virtutech has adapted when developing a new model is to work in an iterative fashion to determine the registers and functions that actually have to be implemented in a device in order to support software by testing the target software with incomplete device models. DML and Simics support a number of techniques for efficiently exploring the needed functionality.

First, a model of the complete register map of a device is created, and registers are marked as “unimplemented” or “dummy” or implemented in a limited fashion. This device model is then used with the software, and any accesses of missing or incomplete functionality is flagged by Simics, neatly pointing out precisely what is still missing in the device model.

Any access to an “unimplemented” register prints a warning message on the Simics command-line. The simulation is allowed to continue, since it is possible that the software is content with a default reply.

“Dummy” registers are registers that the software is using but where the values written can be ignored and any reads return a fixed value (or the latest value written). A typical example is an error counter in a network device—if errors are not modeled, the error counter can be implemented as always reading zero.

For functions which turn out to be needed, starting with only a single mode or a subset of the device functionality, and warning when the software moves outside this envelope, is preferred. For example, a timer might initially only support the simple count-down mode required to trigger periodic interrupts to the operating system, and later adding event-counting functions and similar advanced functionality.

Another technique is to hard-wire the results of reading or writing certain registers to an acceptable result. This is typically done based on the observed behavior of the software, providing values that the software likes. Unlike a dummy register, the eventual goal here is to implement the real functionality of the register. The hard-wired results are just used early in development. The logging facilities of DML allow such hard-wired results to be easily located later and upgraded to real implementations.

The goal is to get the target software up and running as soon as possible, so that problems can be found and the risk of development reduced.

## ASSEMBLE AND TEST THE SYSTEM

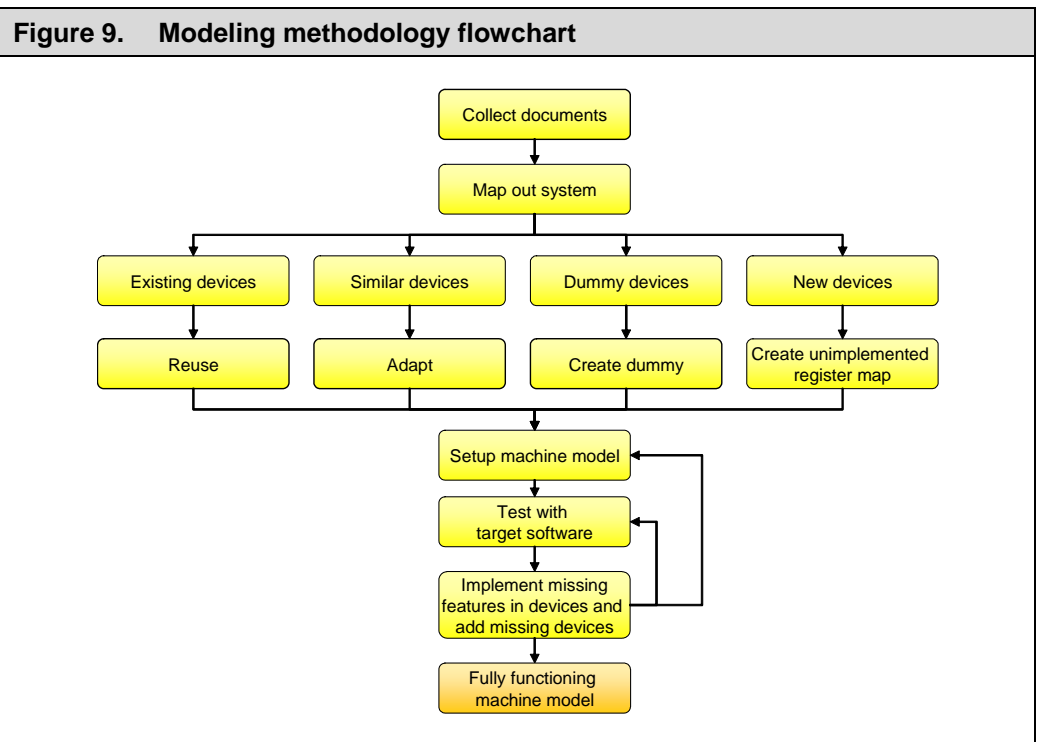
As with devices, a system should be created and composed in an iterative fashion. The goal is to quickly create a minimal system on which the software can run, so that device testing with the real target software can begin as soon as possible. When developing a model of a large system, this usually means following the boot order.

If the target software can be reconfigured to use a subset of the devices needed in the real system, it is best to start with the immediately available devices and processors. The target system is created with a memory map containing the devices which are available. The target software is configured to use only the devices present, and is recompiled. Missing devices are then added as they become available, and iterative development with DML applied to each device in turn. Typically, a minimally useful configuration to use as a starting point contains a processor, some memory from which to execute code, a timer to let an operating system obtain periodic interrupts, and a serial port so that output (and maybe input) is possible. The timer functionality is integrated in certain processor families like PowerPC and MIPS, making the minimal system even simpler.

Another method is to use an existing system which is similar to the new system, and then replace and add components as they become available. This also requires the software to be somewhat flexible about the target hardware on which it runs. It is the natural strategy when developing models of new generations of systems which have already been modeled with Simics, and where the hardware and software are upgraded in parallel. The initial memory map of the system may be different from the target, but as work progresses the memory map will converge to the target machine.

If the software exists and cannot be configured to use a similar system or a subset of system devices, it will be necessary to create initial models of all devices in the system before attempting to run the software. Here, iterative development will be applied to all devices in parallel and the real target memory map will be used from the start.

Figure 6 shows a flow chart for the iterative modeling methodology we propose in this paper.



### **Unit test software**

In addition to testing devices within the target environment with target software, developers often setup unit tests for individual devices. Such tests can be small programs running on the target system processor, or setups using Simics scripting which do not run any code at all.

Unit testing is helpful to isolate and investigate tricky areas of model behavior, and to debug and perform regression tests on the models themselves without worrying about possible changes in the software stack.

For devices which are new, it is often the only kind of testing possible. In this case, it is very advantageous to have a separate programmer create the test cases from the one creating the model. This provides a second interpretation of the specification or documentation for the device, which is a proven method to locate unclear or incomplete details.

## DML and SystemC

DML has been designed specifically to support the efficient definition of device models for virtual software development. As such, the language offers syntax convenience and expressive power particular to the needs of transaction-level device modeling.

SystemC is a language designed to support hardware design broadly. It is a variant of C++, providing access to a simulation kernel and language features supporting modeling of hardware structure and behavior. The base SystemC language is designed for signal-level simulation and interconnection between models.

The OSCI SystemC TLM2 standard provides support for TLM in SystemC. It defines several coding styles or abstraction levels for different use cases. The SystemC LT level of abstraction is very similar to the default Simics level.

DML is a higher-level approach to the modeling problem, providing the benefits of a domain-specific language for modeling efficiency. DML provides support for efficiently describing transaction-level behavior that is not present in SystemC TLM2 or C/C++ with the Simics API.

DML is designed as a productivity solution for transaction-level modeling, regardless of the simulation platform. It is currently implemented for Simics, but the concepts embodied in DML are general and apply to all TLM systems.

## DML, THE DEVICE MODELING LANGUAGE

The most visible part of DML is the specification of the programming register interface of a device. This interface is described as a series of register banks, each of which can be mapped at a different location in the memory or I/O space of a processor. Inside each bank, registers are specified with their size and offset from the start of the bank. A register can be from one to eight bytes, and may be divided into fields consisting of one to sixty-four bits.

DML explicitly specifies the endianness of a register bank, and insulates the programmer from the relationship between the host and target endianness. The same model will run on both little-endian and big-endian hosts without modification, with the same behavior and the same bytes being put in the same places in simulated memory. However, if the target system changes the processor type to a different endianness, the model will exhibit the same endianness issues as the hardware implementation.

Bitfields can be specified to use big-endian or little-endian bit numbering, allowing the specification of fields to follow the programmer's manual for a device regardless of the bit-ordering convention used by the device supplier. In particular, most Power Architecture-related device manuals use big-endian bit order while devices used with x86 and most other processors use little-endian bit order.

Simics allows you to create multiple copies of a device mapped at different locations. The base address of a bank of registers in the processor memory map is specified in the machine setup, and the offset of each register inside the bank is given in the DML file. Thus, DML devices are totally position-independent.

DML makes it easy to make full use of the power of the Simics simulation framework, exposing features such as checkpointing, self-documentation, attributes, event posting, and logging using simple yet powerful language constructs.

To cut down on repetitive coding of registers and other functions within a device, DML provides a template feature that allows bundles of functionality to be reused. Such templates can describe the behavior of a register, field, or other part of a DML model.

Interfaces to other devices and back ends in Simics and DML are described as sets of function calls. A set of related function calls for handling one direction of a model-model interface are grouped into a single named interface data structure that is used when connecting devices. Incoming connections in DML declare which interface they implement, and then implement all the functions in the interface. Outgoing connections specify the interface to use along with a name for the connection. The DML compiler generates the plumbing code needed to let the Simics configuration system connect devices to each other.

The functionality of a model is programmed as small sequential snippets of code expressed in C-style syntax. Any programmer familiar with a language like C, C++, SystemC, Java, or Python can quickly get started writing device models. There are some additional language features to support device modeling tasks like parsing data structure layouts in memory and handling bit slices. The DML compiler automatically generates the code required to activate each sequential snippet, removing the complexity of sequencing and reacting at the right time from the model source code.

Virtutech provides a set of templates and base files to simplify the creation of device models attached to interfaces such as PCI, PCI Express, USB, I<sup>2</sup>C, Ethernet networks, and serial lines. There are also several example devices provided with Simics, and usually the device source code for a target is provided to users of that target.

DML models can also include kernels of code written in C and C++, making it easy to include existing implementations of device functions or algorithms implemented into DML models<sup>4</sup>.

For more details on DML and how the language looks, please see the [White Paper on DML found on the Virtutech website](#).

---

<sup>4</sup> For an example of how to move functionality from software into a hardware accelerator, see the [Virtutech White Paper on System Architecture and Exploration](#).

## SUMMARY

Virtutech Simics enables virtualized software development by providing fast simulated targets that are used instead of physical hardware targets for software development. Simics is a transaction-oriented simulator applying transaction-level modeling for all communication between devices.

To reap the benefits from virtualized software development, Simics users need to construct models of their hardware. This paper has discussed the methods and supporting tools that Virtutech provides to make this process fast and efficient. You can reuse existing models from the Virtutech library, as well as models in C, C++, or SystemC, including custom processor models. With the DML language, efficiently writing new fast transaction-level models has never been easier.

## CONTACT INFORMATION

<b>North and South America</b> sales_americas@virtutech.com	<b>Europe, Middle-East, Africa</b> sales_emea@virtutech.com
<b>Asia-Pacific</b> sales_apac@virtutech.com	<b>Japan</b> sales_japan@virtutech.com
<a href="http://www.virtutech.com">http://www.virtutech.com</a>	

© Copyright 2008 Virtutech, Inc. All Rights Reserved.

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

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

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

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

## CONTACT INFORMATION

<b>North and South America</b> sales_americas@virtutech.com	<b>Europe, Middle-East, Africa</b> sales_emea@virtutech.com
<b>Asia-Pacific</b> sales_apac@virtutech.com	<b>Japan</b> sales_japan@virtutech.com
<a href="http://www.virtutech.com">http://www.virtutech.com</a>	

© Copyright 2009 Virtutech, Inc. All Rights Reserved.

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

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

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

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