Intel® Simics® Simulator Internals Training

Lab 01-06

Communication between Models

1 Introduction

In this lab you will look closely at interfaces between models and how they operate. You will connect two devices through an interface and trigger interface calls directly from the CLI. You will also look at the instrumentation framework as another way to enable model-to-model communication.

2 A Closer Look at Interfaces

Interfaces are essentially collections of functions that a device of class A can expose such that a device of class B can call them without knowing any details about class A. With the interoperability layer of Simics being the C-ABI, regardless of the actual programming language used to implement a device (C, C++, or Python), an interface is defined as a structure of function pointers. This structure determines the function signatures, and the caller only needs a reference to an instance of the structure where the callee has set all function pointers to point to its respective function implementations. So the only thing known to both device classes at compile time is the interface structure and the device classes are entirely independent of each other.

2.1 Inspecting Interface structs

1. Start a new simulation:

```
$ ./simics
```

2. On the CLI, display the **api-help** for the **serial_device** interface. If you know an interface name, you can construct the **struct** name by appending **_interface_t** to it. After typing the first few characters you can also use tab-completion to help you get to the full name.

```
simics> api-help serial_device_interface_t
```

   The response will show you which files you need to include or import when you want to use the interface, the interface structure containing the function pointers and the signatures they expect as well as whether the interface is available in Python.

   NOTE: api-help only works on built-in Simics interfaces. If you are facing a custom interface that is defined in a loadable module, such help will not be available. In such a case, you need to inspect the structure (if needed) through source code.

2.2 Invoking an Interface Method from CLI

In some cases, usually during testing or when debugging, you want to call interface methods directly on object. Let's see how to do that from the Simics CLI.

3. Instantiate an implementer of the **serial_device** interface with the below commands.

```
simics> load-module txt_console_comp
simics> new-txt-console-comp Bob
```

   By this, you get a text console component named "Bob".

4. Inspect the help for Bob's console device called "con".

```
simics> help Bob.con
```

   Amongst other interfaces, the "Interfaces Implemented" section will show the **serial_device** interface being available from the console device.

5. Write a character to the console as shown below:

```
simics> @conf.Bob.con.iface.serial_device.write(ord('!'))
```

You will notice that the exclamation mark appears on the console. You have just invoked the **write** function of the **serial device** interface on the console device! Note that at this point the device is entirely unconnected, but calling into the device is already possible.

If you scroll upwards in the Simics CLI to where you got the **api-help** for the **serial device** interface, you might wonder why the signature of **write** shows a **conf_object_t*** and an **int** as arguments and we only provided an **int**. The reason for this is that whenever interface functions are invoked on objects (for example, when modeling in Python or C++, or when using Python in the Simics CLI), the object instance is passed automatically as the first argument. Only in cases where there is no object to call on (like when modelling in C) the first argument needs to be used.

6. To stress the above statement, invoke the Python help on the **write** function.

```
simics> @help(conf.Bob.con.iface.serial_device.write)
Help on built-in function int (*)([conf_object_t *NOTNULL,] int):

int (*)([conf_object_t *NOTNULL,] int) = <built-in method int
(*)([conf_object_t *NOTNULL,] int) of interface method object>
```

You will note that the first argument is shown as if it was optional, because it is wrapped into square brackets. This indicates that this is the object that this function is called on and will be automatically inserted by Simics in object-oriented contexts.

2.3 Connecting Objects Manually

When using platform models from packages, object connections are taken care of by components, blueprints, or connectors. You rarely need to manually connect objects in running Simics sessions. However, during model development, testing, debugging or maybe when you want to inject faults or experiment with a different connectivity matrix, you'll have to do it, so let's see how that is done.

7. First, run a Python script that defines some classes and creates some objects that we will need in later steps.

```
simics> run-script "%simics%/targets/simics-internals-training/06-
interfaces.py"
```

8. Create an instance of a simple character sending device created just for this training.

```
simics> @SIM_create_object('sender','Alice', queue = conf.clk)
```

This creates and object of class **sender** named **Alice** and its **queue** (the device holding the event queues that can drive virtual time forward) is set to point to an object named **clk** that was created in the Python script you ran in the previous step.

9. Inspect the **Alice'** attribute **to_receiver**.

```
simics> help Alice->to_receiver
```

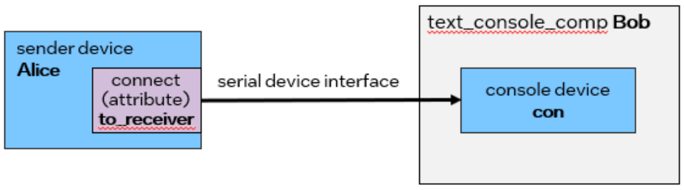
This is a "connect", which is essentially an object type attribute. From that object the interface will be extracted. Note that the mention of the serial device interface is merely in the description text. If the modeler did not add this to the description or attribute name then there is no way of telling what expectations on interfaces such attributes have.

10. Now connect Alice' outgoing connect **to_receiver** to Bob's console device.

```
simics> @conf.Alice.to_receiver = conf.Bob.con
```

Here we use the conf namespace in Python to access the Simics objects. And since we are using Python, object attributes are accessed via dots (".").

Now Alice can invoke **serial device** interface functions on the **to_receiver** connect and they will reach the **con** device. Below is a depiction of the setup.



11. You can verify what a connect is connected to by just inspecting its value.

```
simics> Alice->to_receiver
"Bob.con"
```

This time we accessed the Alice object directly on CLI, so we used the arrow ("->") operator to get the attribute value.

2.4 Observing Interface Calls in Action

Previously we called the write function of the serial device interface manually. Now this time we want to have Alice do the call. The observability of this always depends on the devices. If the caller isn't logging the call and the callee isn't logging anything within the implementation of the interface function, then there will be nothing to observe. In our example here, however, we will be able to see this.

12. The sender class has a string attribute that, when written with a single character string, will emit said character over the **to_receiver** connect after a few cycles. Tell Alice to send the letter "a" to Bob.

```
simics> Alice->char_to_send = a
[Alice info] {clk 0} Will send in 100 cycles.
```

You see that Alice will send in 100 cycles and that the current clock cycle is 0. So we should see the interface function call at cycle 100.

13. Run the simulation 50 cycles forward to have some time pass but not yet reach the point where Alice will send.

```
simics> run-cycles 50
```

14. Now tell Alice to send the character "b".

```
simics> Alice->char_to_send = b
```

```
[Alice info] {clk 50} Will send in 100 cycles.
```

You see that we are at clock cycle 50 and the delay is still 100 cycles, so the letter "b" should be sent at cycle 150.

15. Run another 50 cycles to get to the point where Alice will send the latter "a".

    ```
    simics> run-cycles 50
    ```

    You will notice no logs. This is meant to show that if the device logs nothing, there is nothing to observe on the CLI (frankly, as we will see soon, the console device logs something, but at a higher log level). Apparently, the sender class only logs when it schedules the send but it does not log the send itself. This is to showcase that to understand what happens at interfaces sometimes requires to look at the past in logs.

    However, we do not need any log here, because we can see the effect of the write call directly in the terminal. It now shows the letter "a" so we know that Alice' call to Bob's console device has worked.

16. Increase the log level on the console device to see even more of what is going on.

    ```
    simics> Bob.con.log-level 4
    ```

17. Now run another 50 cycles to get to the point where Alice will send the letter "b".

    ```
    simics> run-cycles 50
    ```

    This time you see the console telling you that it received a char.

2.5 Bidirectional Communication

So far we have looked at unidirectional communication. The sender was simply sending a character to the receiver. In case of bidirectional communication, you usually need two unidirectional interfaces. One from A to B and the other from B to A. We will now look at this, again using the serial device interface.

18. Remind yourself of what the serial device interface entails by using the command below.

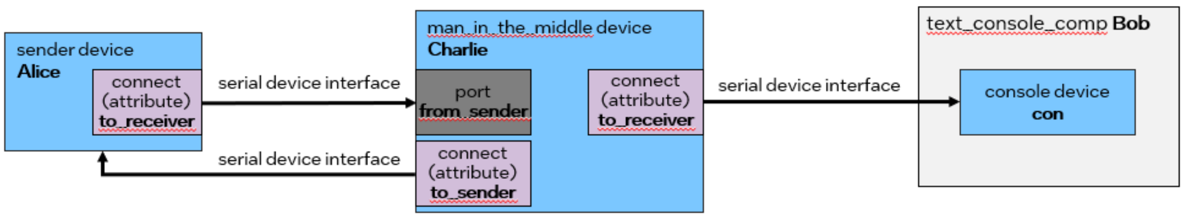
    ```
    simics> api-help serial_device_interface_t
    ```

    In addition to the **write** function you also see the **receive_ready** function. When is this used? If you look at the **write** function you'll see that this has a **non-void** return value: The return value signals how many bytes of the given value have been accepted. This is already a sort of bidirectionality (the receiver can signal back to the sender), but such an approach can only be used for immediate responses that are available at the time of the call. But what if the response is delayed? Imagine the receiver responds with 0 (not accepting the bytes for some reason). How can it tell the sender that it is now okay to send? To facilitate this the sender must also have an interface the receiver can call. In the case of the serial device interface the idea is that both implement the same interface, so both can send bytes and tell each other that they are ready to receive. Such an approach is a symmetrical bidirectional interface. But there are others where the two devices implement entirely different interfaces.

19. Let's insert a device between Alice and Bob that delays the data delivery but cannot accept any additional character while delaying.

```
simics> @SIM_create_object('man_in_the_middle','Charlie', queue =
conf.clk)
simics> @conf.Alice.to_receiver = conf.Charlie.port.from_sender
simics> @conf.Charlie.to_sender = conf.Alice
simics> @conf.Charlie.to_receiver = conf.Bob.con
```

Now Alice sends to Charlie, and Charlie will then send to Bob. Note that Charlie can talk to Alice through its **to_sender** connect. Also note that Charlie implements the **serial device** interface in a port named **from_sender**, while Alice implements it directly at the device level (just like Bob's console device). Below, the new setup is depicted.



20. Now tell Alice to send the letter "c".

```
simics> Alice->char_to_send = c
[Alice info] {clk 150} Will send in 100 cycles.
```

So Alice will send the character at 250 clock cycles.

21. Now run 50 cycles forward half way through the time before Alice will send.

```
simics> run-cycles 50
```

22. Now tell Alice to send the character d in 100 cycles.

```
simics> Alice->char_to_send = d
[Alice info] {clk 200} Will send in 100 cycles.
```

23. Run another 50 cycles to get to the point where Alice sends the character "c".

```
simics> run-cycles 50
[Charlie info] {clk 250} Will forward in 100 cycles.
```

You'll see that Charlie reports that it will forward in another 100 cycles. This, however, means that Charlie will not accept another character for the next 100 cycles, but we know Alice will attempt to send "d" 50 cycles from now.

24. Run 50 cycles more to see what happens when Alice tries to send to Charlie.

```
simics> run-cycles 50
[Charlie info] {clk 300} Currently busy. Cannot forward. Need to wait
until other char is out.
```

```
[Alice info] {clk 300} No bytes written. Waiting for receiver to become
ready.
```

You can see that Charlie is indeed busy and cannot accept the character being sent. You also see that Alice reports Charlies refusal, as it saw the return value from the write call being zero. Now Alice will wait for Charlie to signal that it can accept new characters.

25. Run another 50 cycles to get to the point where Charlie will send the character.

```
simics> run-cycles 50
[Bob.con info] {clk 350} Set char 0x63 at (0, 2) colours 0x100 0x101
[Alice info] {clk 350} Receiver became ready. Re-sending.
[Charlie info] {clk 350} Will forward in 100 cycles.
```

You first see that Bob reports the character arrived, which means Charlie did the delayed delivery. You can also see the character "c" now being shown in the terminal. You also see that Alice reports that the receiver (Charlie in this case) is now ready again for receiving characters, which means Charlie called **`receive_ready`** towards Alice. Finally, Charlie reports that it got yet another character that will be delivered in 100 cycles. So Alice must have called **`write`** towards Charlie again.

As you can see again, observing interface calls is indirect in the sense that you see the effects of the interface function calls, but you do not directly observe the calls themselves.

26. Finally, run another 100 cycles to see the last character being displayed on the terminal.

```
simics> run-cycles 100
```

You see Bob reporting it got the character and you can also see the character on the terminal.

27. This concludes the lab and you can exit the simulation session.

```
simics> exit
```

3 Using the Instrumentation Framework for Model-to-Model Communication

Ultimately all communication between objects in Simics goes through interfaces, be that model interfaces that correspond to hardware interfaces, as we had seen in the previous section, or be that simulator interfaces like attribute set functions. On top of these plain interfaces there can be rules and regulations how to use them and the Instrumentation Framework is such a thing. Ultimately, it uses Simics interfaces, but the Framework dictates a set of required interfaces on instrumentation providers, tools and filters and how they interact. This allows it to integrate more conveniently into the UI and gives a more concise user experience (compared to fully custom interfaces).

The Instrumentation Framework is mainly meant to instrument the simulation to get more insights and to allow deeper analysis, but it can also be used to model behavior, specifically if it is behavior that can be "bolted on top" of an instantiated model. An example of this is the simple cache model that ships with Simics and we will take a closer look at this now.

3.1 Creating a Model Extension with the Instrumentation Framework

1. Start a simulation session running the QSP-x86 based training setup.

   ```
   $ ./simics simics-user-training/001-qsp-training
   ```

2. Once the session is up, check what instrumentation tools you have.

   ```
   simics> list-instrumentation-tools
   ```

   You see that almost all of them for analysis purposes, but you can also see the **simple_cache_tool** amongst them. Also note that you see the associated creation command.

3. To allow the cache model to influence the timing, it needs a helper object of type **cycle_staller**. So let's create that first.

   ```
   simics> new-cycle-staller name = staller stall-interval = 1
   ```

   Note that we use a stall-interval of 1. That means the staller will apply the accumulated stall times (from the various cache layers) every cycle. This is not the normal way to do it. Usually it is better to accumulate over a longer time, but for the purpose of showing things, we apply the delay on every cycle.

4. Now we can create the simple cache tool.

   ```
   simics> new-simple-cache-tool name = cache_tool providers =
   "machine.mb.cpu0.core[0][0]" cycle-staller = staller
   ```

   As mentioned before, even though the Instrumentation Framework uses interfaces to register tools via connections with providers, all this si made more convenient due to the tool creation commands offered by the framework.

5. Now list the active instrumentations.

```
simics> list-instrumentation
```

You see that the cache tool is connected to core[0][0] of the QSP.

6. The cache tool can now be used to "spawn" the actual caches and their hierarchy.

```
simics> cache_tool.add-l1d-cache name = l1d line-size = 64 sets = 64
ways = 12 prefetch-additional = 1 read-penalty = 100
simics> cache_tool.add-l2-cache -prefetch-adjacent name = l2 line-size
= 64 sets = 1024 ways = 20 prefetch-additional = 4 read-miss-penalty =
10000 read-penalty = 1000
```

We will not go into the details of the above setting. The key point is that we have a two-level data-only cache hierarchy that will only impose timing penalties on reads and read misses. Data writes are still tracked, but of no significance to this example. We want to show how such a tool can generally become part of the modelled behavior, not look at what is modelled there.

7. The commands above added new elements to our model. Let's have a look.

```
simics> list-objects -tree namespace = machine.mb.cpu0
┐
[…]
├ cache[0] ┐
│          ├ l1d
│          └ l2
├ core[0] ┐
│         └ [0]
[…]
```

We see the two cache layers we just added.

3.2 Observing the Effects of Model to Model Communication when using the Instrumentation Framework

Like using plain interfaces, there is no direct way to observe that actual interface calls. Again, we can only observe the reaction to the interface function calls. In the case of the Instrumentation Framework there will now be calls originating in the instrumentation providers going via instrumentation connection to the instrumentation tool.

8. We need to run forward to a cacheable instruction. To get there fast, we disable the instrumentation for now.

```
simics> disable-instrumentation name = cache_tool
```

9. Early BIOS instructions are usually not cacheable, so we want to progress into the Linux Kernel. Hence, we run forward to an instruction that we know will do a memory access.

```
simics> bp.memory.run-until 0xffffffff81000148
```

10. Now enable the instrumentation again.

```
simics> enable-instrumentation name = cache_tool
```

11. Display the current time.

```
simics> ptime
```

Remember the steps and cycles.

12. Now increase the log level on our cache hierarchy as we expect this to be active.

```
simics> machine.mb.cpu0.cache[0].log-level 4 -r
[machine.mb.cpu0.cache[0]] Changing log level recursively: 1 -> 4
```

13. Disassemble the current instruction.

```
simics> da
cs:0xffffffff81000148 p:0x000200148  mov rsp,qword ptr [rax+0xa18]
```

You see that this instruction will read a quad word from memory, so we can expect our data cache to have an opinion about that.

14. Progress the simulation by instruction.

```
simics> run 1
```

You see responses from the cache models. L1 is missing and hence asking L2. L2 is missing for the direct access and the L2 prefetches, but the subsequent L1 prefetch misses in L1 but hits in L2 (since it prefetched already).

15. Check the current time again.

```
simics> ptime
```

You will notice that the step counter increased by 1, because one instruction was executed. But you also see the cycles having increased by 11,101 cycles. This is one cycle for the instruction, 100 cycles for the general L1 read penalty, 1000 cycles for the general L2 read penalty and 10000 cycles for L2 miss penalty (which is applied once per instruction and not once per missing fetch).

So you could see that through the Instrumentation Framework the model could be extended and afterwards the model objects were communicating to model both functional behavior (cache state) and timing behavior (hit/miss penalties).

16. This concludes the lab. Exit the simulation session.

```
simics> exit
```