Simics Cheatsheet

1 Simics Usage (command line)

Glossary & Documentation

Simics *Base*	simics executable + the set of supporting shared libraries (DLLs) + tools
simulation	running of code (target) on a model/platform (target) with advancement of time
host	a computer where simulation runs
target	a simulated code running in its isolated memory region, e.g. Linux or Windows guest
platform	a complete runnable model: full set-up of devices with CPU or with at least a clock provider
package	a set of devices, oftentimes constituting 1 platform, distributed as a whole in an archive and unpacked into 1 directory
[Simics] Script	simple Unix-shell-like language (a wrapper over Python) used for connecting devices and command line automation
Simics User's Guide	documentation on **Simics** usage — both command line and Eclipse

Below **$** stands for Unix shell prompt, **>** for **Simics** prompt. *Big_italic* text means user-supplied arguments for commands/functions.

Set up a workspace with platform package

```
$ path/to/simics-base/bin/project-setup .
$ bin/addon-manager -c # remove default package associations
# Allow scripts & shared libraries be found:
$ bin/addon-manager -s /path/to/platform-package
$ bin/addon-manager -s /path/to/additional/package
```

Start up

```
$ ./simics targets/platform/platform.simics
```

Shell command line arguments ⟶ **Simics** command line correspondence:

```
$ ./simics start.simics
```
⟶
```
$ ./simics
> run-command-file start.simics
```

```
$ ./simics script.py
```
⟶
```
$ ./simics
> run-python-file script.py
```

```
$ ./simics -e '$config_variable1=value; $config_var2=value' start.simics
# ⚠ NOT ./simics start.simics -e $varibables ...
```
⟶
```
$ ./simics
> $config_variable=value
> $config_var2=value
> run-command-file start.simics
```

Environment & Packages

```
> version # list of installed packages
$ ./simics -v # the same
> pwd # current directory where simics is running
> list-directories # where Simics searches files
```

To debug chain of called auxiliary scripts include s:

```
$ ./simics -script-trace targets/platform/platform.simics
```

Commands for running simulation

```
> c[ontinue]
> r[un] 100 cycles # or 10 steps or 0.1 seconds
> ptime [-all] # to show target's time
> print-event-queue # show all pending events (e.g. timers)
# To know CPU/clock which advances time for the device (and executes events):
> platform.myDevice->queue
```

Printing device structure

To find devices by name/class or interface name:

```
> list-objects -all name # searches also for the class
> list-objects -all substr = mem # object names containing mem
> list-objects -all iface = my_interface # by interface name
```

To examine device structure (**Simics** *components* and *devices*) of a given component *myPlatform*:

```
# 1-level representation: immediate children myPlatform:
> list-objects namespace = myPlatform
# Multi-level one: all children of myPlatform with all hierarchy:
> list-objects namespace = myPlatform -tree
```

To find all objects with the same class as given device:

```
> platform.myDevice->classname
myClass
> list-objects -all myClass
> list-objects -all class = myClass
```

Registers

```
# Print all registers:
> print-device-regs platform.myDevice
# Print fields of register myRegister:
> print-device-reg-info ↩
  platform.myDevice.myBank.myRegister
```

Writing/reading with side effects

```
> write-device-reg ↩
  platform.myDevice.bank.Bank.Group.myRegister 0x1
# (register groups like myGrp may be omitted in devices)
> read-device-reg ↩
  platform.myDevice.bank.Bank.Group.myRegister
```

Writing/reading **without** side effects (aka set/get)

```
# You can use set-device-reg / get-device-reg...
# ...or do it through attributes, e.g. to set to value 0x1:
platform.myDevice->Bank_Group_register = 0x1
# To get the value:
platform.myDevice->Bank_Group_register
```

Connects — attributes that point to other devices

```
# To set a connect attribute:
> platform.myDevice->myConnect = "platform.Device2"
# To zero connect attribute:
> platform.myDevice->myConnect = FALSE # obvious ;-)
```

Device information: static

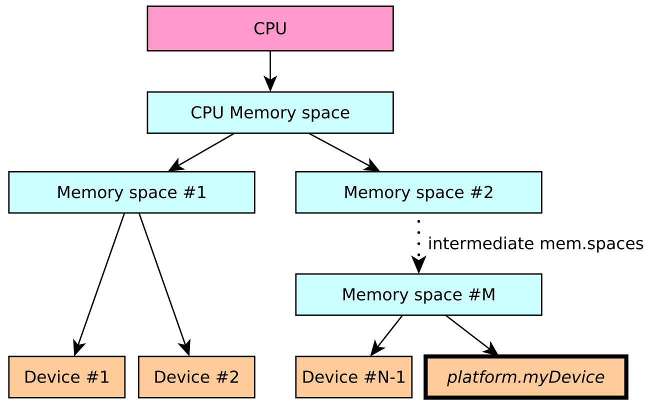
```
# To find info about class/module/package for your device:
> help platform.myDevice
  Class myClass
  Provided By
    myModule (from myPackage)
  # ...then documentation about the device is printed.
# To list all classes provided by a module:
> list-classes -m myModule
# Configuration information:
> platform.myDevice.info
```

Device information: dynamic

```
# Runtime information:
> platform.myDevice.status
# pretty-print device attributes with values:
> list-attributes platform.myDevice
# to search all attributes/registers containing mem:
list-attributes platform.myDevice substr = mem
```

Memory spaces

All target software runs on CPU and can access devices only through hierarchy of memory spaces:




```
# To find memory spaces the device is mapped in:
> devs platform.myDevice
```

Getting help

```
# Get help on some command:
> help instantiate-components
# Finding sections of documentation mentioning it:
> apropos instantiate-components
# To open Simics Documentation in browser run from workspace:
$ ./documentation # or .\documentation.bat on Windows
```

Debugger commands

To use **Simics** to debug target:

```
dis[assemble] # show assembler commands at current address
# Set break point on physical memory access from current CPU:
break address [-r] [-w] [-x] # -rw by default
# To break from specific CPU:
cpu.phys_mem.break address [-r] [-w] [-x]
# To break on access from any memory space to myDevice:
break-io myDevice Bank Offset Length
# To break on specific conditions, "HAP"s:
break-hap Core-Magic-Instruction # run list-haps for options
```

Examining current state

pselect	show currently running CPU
memory-map	print all mapped devices
pregs	print all CPU registers
probe-address *addr*	path to target *addr*

x86-specific and platform-specific commands

memory-trace *addr*	path to target *addr*
pregs	to know x86 mode (16/32/64-bit), 1st line
reset-button-press	to reboot the target
power-button-press	to press power button

Moving files target ⟷ host

```
> start-simicsfs-server
```

```
target$ mkdir a
target$ simicsfs-client a
```

Saving info

Save logs from current point

```
> start-command-line-capture filename
```

To save simics variables to a file:

```
> start-command-line-capture filename
> list-variables
> stop-command-line-capture
```

Check points

write-configuration "*checkpoint_name*"	— save checkpoint
read-configuration "*checkpoint_name*"	— restore it back

Creating components dynamically (for hot plug)

```
> create-myDevice-comp "platform.myDeviceComp"
> connect platform.myDeviceComp.connectorName ↩
   platform.Device2.connectorName
> instantiate-components
```

Miscellaneous

```
# To dump network packages (for analysis with Wireshark):
> pcap-dump link=ethernet_switch0 filename=myFile
```

```
# Treat all input as Python code:
> python-mode
> > > Enter your Python code
```

Glossary & Documentation

DML	Device Modelling Language is a wrapper over C for writing (fast) devices. Its compiler is included into **Simics** Base.
device	any **Simics** class for modelling real devices. Different **Simics** devices communicate via *interfaces*.
Model Builder User's Guide	overview of **Simics**/DML programming
DML 1.4 Reference Manual	language specification
API Reference Manual	API function list for DML & C, describes ownership rules
model writers	those who write **Simics** devices
users	those who use **Simics** devices: device driver writers, firmware/UEFI writers, validators/testers, etc

Getting help

```
# Print explanation of C/Python/DML API functions:
> api-help SIM_add_configuration
```

Create stub device

```
$ bin/project-setup --device example-dev
$ ls modules/example-dev
test/ example-dev.dml Makefile module_load.py
```

The header of *example-dev*.dml is then:

```
dml 1.4;               // obligatory .dml header
device example_dev;    // class name.
                       // Note: - changed to underscore _
```

Create stub interface with Python wrapper

```
$ bin/project-setup --interface sample
$ ls modules/sample-interface
Makefile sample-interface.dml sample-interface.h
```

Python support is enabled by IFACE_FILES in the Makefile. The generated C **struct** name is *sample_interface_t*.

Arithmetics

RHS is int64, LHS truncates

Assignments are equivalent to casts and hence can truncate:

```
local uint8 x = 0xffff // results in x == 0xff
```

All aritmetic operators like $+$, $*$ convert its operands to int64:

```
local uint16 i = 0x7fff; local int8 j = 2; // Let us sum them:
local int16 x = i+j;   // i+j calculates as int64 → 0x8001, truncates to int16
// finally results in x == 1.
```

Comparisons act as uint64 or int64: ==, <, <=

Comparisons on only **uint64** act as proper **uint64**, however in comparisons **int64** vs. **uint64** the operands are converted to **int64**!

```
local uint64 u;
u = -1;     // equivalent to u = cast(-1, uint64) = 2^64 − 1, all ones
u == -1     // FALSE! Equivalent to int64(u) == int64(-1),
            // where upper bit is cropped: int64(u) == (2^63 − 1).
u == cast(-1, uint64) // true. As comparison is b/w two uint64
u > -1      // true, but unlike C! it's int64(u) > int64(-1): 2^63 − 1 > 1
```

Syntax

Statements

```
// Printing through log statements:
log log-type, level, groups: "format-string", arg_1, ..., arg_N;
                (default 1) (default 0 (no group))
// (the "format-string" is the same as in C, see 'man 3 printf')
```

log-level	usage rule
1	most important messages (for both users and model writers), typically `error`
2	crucial events for boards/devices, e.g. their resets
3	any other messages (for users)
4	internal device debug messages (for model writers)

log-type	usage rule
`info`	informational message
`spec_viol`	specification violation by target software (for users)
`unimpl`	attempt to use not implemented functionality (for users)
`error`	internal device error (for model writers). ⚠ There is a limit (default 10_000) after which simulation stops!
`critical`	like `spec_viol` or error, but ⚠ stops simulation

```
// Dynamic allocation (like malloc() in C):
local type * x = new type;
// E.g. for int array:
local int * x = new int[100];
delete x; // deallocation like free() in C
// Raising/catching exceptions:
try {
    throw; // YES, no data can be carried by exception
} catch {
    ...
}
```

Expressions

```
sizeof value // : int — get byte size of the value
sizeoftype type // : int — get byte size of the type
x[10:8] // get bits 10—8 of integer x:
x[8] // get bit 8 of integer x:
```

Scalar types

DML-specific: **uint1**...**uint64** and **int1**...**int64** (+ aliases: **int** → **int32**, **char** → **int8**), **uint8_be/le_t** ... **uint64_be/le_t**.
C-like types: **size_t**, **uintptr_t**, **double**, **bool**.

Derived types

```
typedef struct { member declarations; } typeName;
// Layout members can be only whole-byte (int8, int16, ...):
typedef layout "big-endian" { member declarations; } typeName;
// Bitfield size sizeBits can be 1 ... 64 and field size arbitrary:
typedef bitfields sizeBits {
    uint3 a @ [31:29];      // just an example
    ...
} typeName;
// All the types can be used for variable definitions inline, e.g.:
{
    local struct { uint8 field; } variableName;
    variableName.field = 255;
```

Methods

They are called *methods*, not functions, because they accept **implicit** 1st argument — object (current device), like C++ methods. Thus multiple instances of each device are allowed.

```
method name(inputType1 arg1) -> (outputType1, outputType2)
{
    ...
    local outputType1 var1 = ...; local outputType2 var2 = ...;
    return (var1,var2);
}
// Calling this method:
local inputType1 val = ...;
local outputType1 x; local outputType2 y;
(x, y) = name(val);
```

Bitfields

```
bitfields 32 {
    uint3 upper_bits @ [31:29];
    ...
}
```

Object declarations

```
objectType objectName {
    method methodName {
        ...
    }
}
register regName @ offset is (template1, ...);
// @ offset is a syntax for "param offset = offset;"
```

Module variables and other data objects

DML construct	check-pointed	fields	address-mapped	arbitrary data
session	-	-	-	+
saved	+	-	-	-
attribute	+	-	-	+
unmapped register	+	+	-	-
[normal] register	+	+	+	-

Interfaces

Definition in `.dml`:

```
extern typedef struct {
    method name(conf_object_t *obj, type1 value1)
        -> out_type;
} sample_interface_t;
```

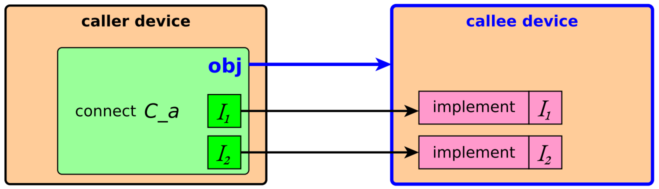
Obligatory definition in C `.h` file **for using the interface from Python**:

```
SIM_INTERFACE(sample) { // essentially 'typedef struct' also
    out_type (*name)(conf_object_t *obj, type1 value1);
}
#define SAMPLE_INTERFACE "sample"     // necessary!
```

Then it is called **without** `obj`: just `name(value1)`.

Explaining `connect`s and `implement`s

- `connect`s are for **out**bound calls
- `implement`s are for **in**bound calls



	unnamed (whole-device)	named (wrapped in attribute)

`connect`	-	`connect C_a {` ❶
		`  interface I1 {` ❷
		`    ...`
		`  }`
		`  interface I2 {` ❷
		`    ...`
		`  }`
		`}`
		`connect C_b {` ❶
		`  interface I2 {` ❷
		`    ...`

`implement`	`implement I1 {`
	`  ...`
	`}`
	`implement I2 {`
	`  ...`
	`}`
	as in picture above
		`port P_A {`
		`  implement I1 {`
		`    ...`
		`  }`
		`  implement I2 {`
		`    ... 1st way of I2 ...`
		`  }`
		`}`
		`port P_B {`
		`  implement I2 {`
		`    ... 2nd way of I2 ...`

To make the whole `connect` required:

❶
```
param configuration = "required";
// other variants: "optional" (default), "pseudo", "none"
```

❷ Individual interfaces are already **required** by default, to make them optional:

```
param required = false;
```

⚠ `implement {}` does NOT check that all (or any!) methods of an interface are really provided, defaulting to **NULL**.

Calling device code from Python

If *device* `implement`s interface $iface_1$ then its methouds can be invoked from **Simics** command line via Python:

```
@conf.platform.device.iface.iface1.method1(argument_1, ...)
```

Debugging with Gdb

To debug simics and its modules itself:

Terminal #1:

```
# Recompile your module with debugging support:
$ make clobber-my-module
$ make D=1 my-module
$ ./simics
> pid
12345
```

Terminal #2:

```
$ bin/gdb
>>> attach 12345
>>> br file.dml:100      # set break point on line 100 of file.dml
>>> continue
```

Back to terminal #1:

```
> run-command-file targets/platform/platform.simics
```

Using gdb for debugging target

```
load-module gdb-remote
new-gdb-remote 50000      # open port 50000
```

Attribute values

`attr_value_t` is a C union that can hold one of a few predefined types. Attributes values are allocated/packed by:

`attr_value_t x = SIM_make_attr_T(cType val)`, and extracted by:

`cType val = SIM_attr_T(x)`, where *T* and *cType* can be:

T	type spec	*cType* — DML/C type
uint64, int64	i	**uint64**, **int64**
boolean	b	**bool**
floating	f	**double**
string	s	**char***
object	o	**conf_object_t**
list	$[x_1 ... x_n]$	fixed-width tuple with n elements of types x_1, ..., x_n
list	$[x*]$	arbitrary-width array of x
list	$[x+]$	non-empty arbitrary-width array of x
list	$[x\{m:n\}]$	array of x with $m \le size \le n$
list	$[x\{n\}]$	fixed-width tuple with n elements of x
dict	D	array of **attr_dict_pair_t**
data	d	**uint8***
nil	n	**void** or x*****
invalid		(none, used for indicating errors)

List items are accessed by `SIM_attr_list_item`. Type specs can be OR'ed as $x_1|x_2$. Type spec is used in `param type = "..."`. For first 4 types there are predefined DML templates `uint64_attr`, `int64_attr`, bool_attr, double_attr.

Attribute initialization

Execution stage	SIM_object_is_configured(obj)	SIM_is_restoring_state()
Create object at 1st platform init	-	-
Load checkpoint	-	+
Load micro-checkpoint (reverse execution)	+	+
Manual attribute assignment (hot plug from **Simics** command line)	+	-

Standard register templates

read, write — make a register readable/writeable, init_val — reset a register to value of `param init_val = ...` of the same name.

Compile-time statements & conditional compilation

```
param p1 = 10; // non-overridable parameter
param p2 default "value"; // an overridable parameter
template myTemplate {
    // Undefined value — must be given
    // on myTemplate instantiation:
    param p3;
    ...
}
// Compile-time if:
#if (p1 == 20) {
    ...
} #else #if {
    ...
} #else {
    ...
}
// Compile-time ternary operator #? #::
param mode = p1 == 20 #? "equal 20" #: "not equal 20";
// Represent value of parameter as string:
param p1_str = stringify(p1); // results in "10"
```

Hash tables

```
import "simics/util/hashtab.dml";
...
local ht_str_table_t tab; // str — string (aka const char*) keys.
ht_init_str_table(&tab, /*keys_owned*/ true);
local double *value = new double; *value = 10.0;
ht_insert_str(&table, "key", cast(value, void *));
local double *get_back = cast(ht_lookup_str(&tab, "key"), double*);
assert *get_back == 10.0;
```

There are also tables for `int` keys or general (`common`) keys.

The secret of DML

Many "internal" features like registers and even connects are actually normal templates for objects (`is object;`) in plain DML defined in `simicsBase/{linux64|win64}/bin/dml/1.4/dml-builtins.dml` and thus they can be expanded.

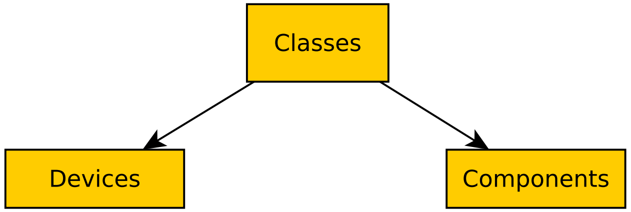
C API

It's possible to do most things in C, e.g. create device by `SIM_create_object`, though normally it's done from Python components.

3 Simics configuration and build system

Glossary & Documentation

Python	this language is used for connecting devices together (writing components), for writing some (slow) devices, for unit-testing
Component	a special **Simics** class that forms a namespace tree and (typically) in its nodes contains instances of device classses. Components implement required `component` interface and optional `component_connector` interface.



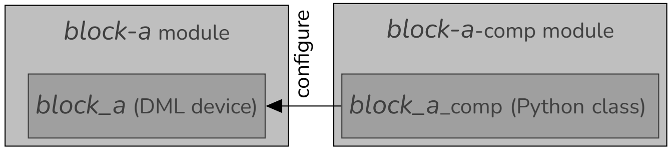
Modules/Components/Classes

A **Simics** module includes classes, there are 2 types of them:

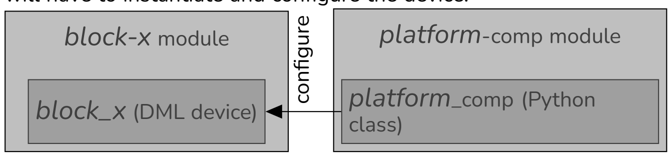
- devices, typically written in DML
- *components*, typically written in Python

Component is a **support** entity: normally components are used only on initialization phase to **configure** devices (set their attributes). During simulation only device instances act (the known exception is hot-plug).

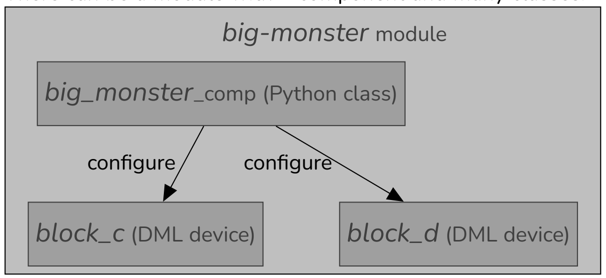
A typical layout with 1-to-1 device-component correspondence:



In simplest case there are no components for a device, so its *platform* will have to instantiate and configure the device:



There can be a module with 1 component and many classes:



Components vs simple devices

Components/connectors are used:

- when there is a need to unite big number of devices to prevent pollution of the surrounding namespace
- when this is a separate device that can be used across different packages independently

Otherwise use simple devices by just assigning *connects* to *implements* directly using **Simics** Script or Python:

```
> platform.device_1->connect_1 = platform.device_2
@conf.platform.device_1.connect_1 = conf.platform.device_2
```

⚠ Do not confuse: *Connectors* connect *component* objects, while *connects* (+implements) connect *device* objects. The `connect` command acts on (component) connectors only!

Creating devices in tests

```
x = simics.pre_conf_object("myDevice", "device_name")
x.some_required_attribute = y
    # (where y can be another pre_conf_object)
simics.SIM_add_configuration([x], None)
```

Creating devices in components

```
class Component_name(Standard_component):
    def add_objects(self):
        x = self.add_pre_obj('myDevice', 'device_name')
        x.some_required_attribute = y
```

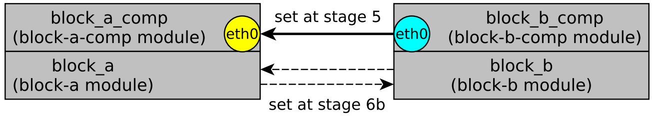
Then slot *myDevice* is created inside any instance of *Component_name*.

Structure and initialization order

A typical structure of component class code:

```
class block_a_comp(StandardComponent):
    @classmethod
    def register(cls):
        ...
    def __init__(self):
        ...
    def setup(self):
        ...              calls
    def add_objects(self):
        ...
    class attribute_name(ConfigAttribute):
        def _initialize(self):
            ...
        def _finalize(self):
            ...
    class component_connector(Interface):
        def get_connect_data(self, block_a_connector):
            ...
        def connect(self, block_a_connector, data_from_b):
            ...
    ...
    class component(StandardComponent.component):
    ...
```

Imagine we want to connect 2 components with their corresponding devices, where block_b has an *Up* connector `eth0` and block_a has a *Down* connector `eth0`:



Let us examine initialization of modules ⟶ components ⟶ devices for this sample program:

1. load-module block-a-comp
2. load-module block-b-comp
3. create-block-a-comp "block_a_component"
4. create-block-b-comp "block_b_component"
5. connect block_a_component.eth0 block_b_component.eth0
6. instantiate-components

The corresponding break down of functions that are normally simics-defined or user-defined.

1

- block-a-comp/module_load.py:__main__
  - block-a-comp/block_a_comp.py: class block_a_comp.register()

2 The same for block_b_comp

3 Note that add_objects adds normally pre-configured objects:

- create object block_a_comp
- block_a_comp.__init__(self)
- block_a_comp._initialize(self)
  - for earch block_a_comp attribute
    - attribute_name._initialize(self)
- block_a_comp._finalize(self)
  - for each block_a_comp attribute
    - attribute_name._finalize(self)
- block_a_comp.setup(self)
  - user-supplied function like add_objects(self)
    - call self.add_pre_obj("block_a_device", "block_a")

4 The same for block_b_comp & block_b

5

- block_a_comp.get_connect_data(self, block_a_connector)
  ... return data_from_a
- block_b_comp.get_connect_data(self, block_b_connector)
  ... return data_from_b

Note get_connect_data is basically just a switch/case statement that chooses which data (object references, port names, etc) to pass to connect function call below.

6a

- block_b_comp.class component_connector(Interface).connect(
  self, block_b_connector, data_from_a)
  - set connects of block_b DML device:
    block_b_component.block_b_device = data_from_a["field"]
- block_a_comp.class component_connector(Interface).connect(
  self, block_a_connector, data_from_b)
  - set connects of block_a DML device:
    block_a_component.block_a_device = data_from_b["field"]
- block_a_comp.component.pre_instantiate(self)
  block_b_comp.component.pre_instantiate(self)

6b Pre-configuration phase ends and finally instantiate-components begins to configure real DML objects:

- block_a: call init() on all attributes with \`init\` template applied.
  block_a: call device method init()
- block_b: call init() on all attributes with \`init\` template applied.
  block_b: call device method init()
- block_a: call set() for all connects/attributes
- block_b: call set() for all connects/attributes
- block_a: call device method post_init()
- block_b: call device method post_init()

After that phase self.get_slot("name") returns real, already configured, device objects.

6c Rarely some additional tweaks are required on configured objects:

- block_a_comp.component.post_instantiate(self)
- block_b_comp.component.post_instantiate(self)