

# Atacama Large Millimeter

ALMA-SW-NNNN

Revision: 6.0

2006-07-26

## A CS Component Simulator - Design and Usage

*Design and Usage Document*

David Fugate

*University of Calgary*

**Keywords: DynamicImplementation, ACS, Component, Container**

Author Signature:

Date:

Approved by:

Signature:

Institute:

Date:

Released by:

Signature:

Institute:

Date:

*Change Record*

REVISION	DATE	AUTHOR	SECTIONS/PAGES AFFECTED
	REMARKS		
3.1	2003-12-11	D. Fugate	All
	Created		
3.1	2003-12-13	D. Fugate	All
	Missing design aspects but complete for the most part.		
3.1.1	2004-02-06	G. Chiozzi	All
	Added comments.		
3.1.1	2004-03-18	D. Fugate	All
	Updated to reflect Gianluca's comments. Still needs UML diagram fixed.		
4.0	2004-09-02	D. Fugate	All
	Added some preliminary instructions on using the simulator.		
4.0	2004-11-09	D. Fugate	All
	Updated for ACS release		
4.0.1	2005-02-10	D. Fugate	CDB
	Added note about CDB usage.		
4.0.1	2005-03-11	D. Fugate	CDB
	Added small section about accessing in/inout parameters.		
5.0.4	2006-07-25	D. Fugate	All
	Partially updated for ACS 5.0. Needs work still.		

## Table of Contents

<b>1 Scope</b> .....	<b>5</b>
<b>2 Overview</b> .....	<b>5</b>
2.1 Abbreviations and Glossary.....	5
2.2 References.....	6
<b>3 Design</b> .....	<b>6</b>
3.1 Requirements.....	6
3.2 Implementation Language.....	7
Class and Data Descriptions.....	7
To be redone for ACS 6.0! .....	7
3.3 UML Diagram.....	7
To be redone for ACS 6.0! .....	7
3.4 Configuration Database Entries Defining IDL Interfaces.....	7
3.5 Graphical User Interface.....	12
To be redone for ACS 6.0! .....	12
<b>4 Usage</b> .....	<b>12</b>
Getting Started.....	12
Editing the CDB to Specify Simulated Components.....	12
Editing the CDB to Specify Default Behavior for a Simulated Component.....	13
Using Interactive Container Sessions.....	17
After defining functions, make sure you add one extra newline after the last line of your function.....	18
When defining functions to simulate component methods using the “setComponentMethod” API function; your function should only accept one parameter regardless of how many in/inout/out parameters the IDL method defines. This parameter corresponds to a list of the real parameters passed to the IDL component. To further illustrate this, see the stopRamping implementation above.....	18
After striking the “Enter” key, all characters on the current line are sent to the real Python interpreter to be evaluated.....	18
While you may be able to “Backspace” your way up to the previous line, this has no real effect.....	18

Messages from the real Python interpreter sent to stderr will end up in red text within the widget. Non-erroneous messages are sent to the real Python console as would occur normally.....	18
Using the GUI.....	19
Special Notes.....	21
Accessing Method Parameters.....	21
<b>5 Known Problems and Issues.....</b>	<b>21</b>
<b>6 Appendix.....</b>	<b>21</b>
RampedPowerSupply IDL Interface.....	21
PowerSupply IDL Interface.....	21

## 1 Scope

This document describes in detail the design and usage of a CORBA object simulator written in Python.

## 2 Overview

Shortly before the release of ACS 3.0, it was decided there was a need for a dynamic component (derived from *ACSCComponent*) capable of implementing its IDL interface at run-time. For those unfamiliar with ACS terminology, *ACSCComponent* is an IDL interface defined in the *acscomponent* module and it is the base interface for all components within ALMA software. Editor's note: as of ACS 4.0, the simulator framework also supports components derived from the BACI *CharacteristicComponent* IDL interface.

A dynamic, "simulated" component becomes especially useful when a developer wants to test Components against an IDL interface that has no concrete implementation yet. This simulator will either generate return values, exceptions, etc. on the fly or accept these parameters from the user in some form or another (i.e., command-line, GUI, configuration database, etc.). Also, the simulator takes advantage of ACS helper classes where applicable (i.e., *ComponentLifecycle*, *ACSCComponentImpl*, etc.).

For those who only want to learn how to use the ACS simulator framework, you can safely skip ahead to the section on usage.

### 2.1 Abbreviations and Glossary

<b>ACS</b>	ALMA Common Software
<b>acspy</b>	ALMA Common Software Python software module
<b>BACI</b>	Basic Access and Control Interfaces
<b>CDB</b>	ACS Configuration Database
<b>characteristics</b>	The run-time properties of an IDL interface's implementation. This includes but is not limited to return values, timeouts, and the native code methods should execute.
<b>component</b>	A CORBA object derived from the <i>ACS::ACSCComponent</i> IDL interface
<b>container</b>	A CORBA object responsible for the lifecycle of components.
<b>CORBA</b>	Common Object Request Broker Architecture
<b>DynamicImplementation</b>	A dynamic implementation class for IDL interfaces.

<b>IDL</b>	Interface Definition Language
<b>MACI</b>	Management and Control Interface

## 2.2 References

- [R01] ACS Architecture (<http://www.eso.org/~gchiozzi/AlmaAcs/OnlineDocs/ACSArchitecture-4.0.pdf>)
- [R02] ACS Software Module: “acspy” (ACS/LGPL/CommonSoftware/acspy in the ALMA CVS repository)
- [R03] Python Essential Reference (Beazley)
- [R04] CORBA ([http://www.omg.org/technology/documents/formal/corba\\_iiop.htm](http://www.omg.org/technology/documents/formal/corba_iiop.htm))
- [R05] OMG IDL-Python Mapping (<http://www.omg.org/technology/documents/formal/python.htm>)
- [R06] ALMA Python Coding Standards  
(<http://www.eso.org/projects/alma/develop/alma-se/reference/CodingStandards.html>)
- [R07] A GENERIC SIMULATOR OF CONTROL SYSTEMS FOR APPLICATION DEVELOPMENT AND TESTING  
([http://kgb.ijs.si/KGB/articles/PCaPAC2002-A\\_Generic\\_Simulator\\_of\\_Control\\_Systems\\_for\\_Application\\_Development\\_and\\_Testing.pdf](http://kgb.ijs.si/KGB/articles/PCaPAC2002-A_Generic_Simulator_of_Control_Systems_for_Application_Development_and_Testing.pdf))
- [R08] BACI Specifications Document  
([http://www.eso.org/~gchiozzi/AlmaAcs/OnlineDocs/ACS\\_Basic\\_Control\\_Interface\\_Specification.pdf](http://www.eso.org/~gchiozzi/AlmaAcs/OnlineDocs/ACS_Basic_Control_Interface_Specification.pdf))

## 3 Design

### 3.1 Requirements

1. Must be able to generate complete implementations of all IDL methods and attributes of an interface on the fly.
2. Enumerations will be fully supported.
3. If an interface defines a CORBA object attribute or a method that returns a reference to another CORBA object, the simulator should create the CORBA object.
4. A simulated component should behave in the same manner as a real component. That is, simulated components shall be derived from the *ComponentLifecycle* class and other utility classes where applicable.

5. Users will have the option to specify a timeout value for methods. When the method is invoked it will sleep for a period of time defined by the timeout and then return.
6. Read/write attributes should have some form of “memory” to store the value if it is being set. If the attribute is being read and it is not in the ‘RANDOM’ state, the “memory” value will be returned.
7. This design recognizes the fact that it may be necessary to simulate the “crashing” of a component.
8. A GUI shall be implemented allowing developers to set return values, timeouts, etc. for each attribute and method an IDL interface defines. Furthermore, the GUI will accept as a parameter a Python package and a function name which can substitute for a simulated attribute or method. If the developer does not set these parameters via the GUI, the infrastructure should then search the ACS configuration database and if the values cannot be found there either, they will be generated on the fly.
9. The GUI will be a dumb client for all intensive purposes. In other words, the intelligence of the simulation will reside in an API available to developers and the GUI will just make requests of the API. This will be used to facilitate the simulator’s use in modular tests in particular.

### **3.2 Implementation Language**

Because the simulator should be able to emulate components where the IDL interfaces are not known ahead of time, a dynamic programming language seems like the logical choice for an implementation. Python is both dynamically scoped and typed, supports dynamic inheritance, and most importantly allows developers to dynamically redefine methods at run-time. ACS already provides a Python container which makes Python the ideal language for the simulator’s implementation.

### **Class and Data Descriptions**

To be redone for ACS 6.0!

Description goes here.

### **3.3 UML Diagram**

To be redone for ACS 6.0!

Description goes here.

### **3.4 Configuration Database Entries Defining IDL Interfaces**

The usefulness of predefining simulated component behavior is acknowledged by this document. For example, perhaps the end-user wants to find out what happens when the return

value of some method is fixed, but non-default. Productivity may be hampered because of the time spent changing return value(s) each time the simulated component is activated. For reasons like this, the characteristics of a simulated component should be retrieved from the ACS configuration database if the user does not explicitly set them by some other means. The CDB entries will be placed in \$ACS\_CDB/alma/simulated/\* in a similar fashion to BACI *CharacteristicComponents*. By doing this, simulated BACI properties will be able to use the real CDB configuration files for the component (if available). The only real difference between the two is that simulated components must validate against the SimulatedComponent.xsd schema defined as follows:

```
<xs:schema xmlns="urn:schemas-cosylab-com:SimulatedComponent:1.0" xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="urn:schemas-cosylab-com:SimulatedComponent:1.0" elementFormDefault="qualified"
attributeFormDefault="unqualified">
  <xs:complexType name="imports" mixed="true">
    <xs:annotation>
      <xs:documentation>
        This end-user nicety allows developers to declare Python import statements
        which will then automatically be imported into the simulator framework.
        For example, this element could contain:
```

```
from time import sleep
import sys
import FRIDGE
```

It must be noted that these lines must not be preceded by white space and are limited to simple import/from statements.

```
</xs:documentation>
</xs:annotation>
</xs:complexType>
<!--#####-->
<xs:complexType name="event" mixed="true">
  <xs:annotation>
    <xs:documentation>
      The event type is an XML element describing ALMA events and channels.
      What this element does is tell the simulator framework that a given event type on a
      given channel should be sent out using:
      1. A block of Python code existing within this element where the last line
      corresponds to an event. An example could be something similar to:
```

```
joe = FRIDGE.temperatureDataEvent(7L)
joe
```

2. A random instance of the event type generated by the simulator framework.

Event also allows end-users to send events on a given frequency. Finally, an attribute exists which allows setting the probability that the event will not be sent at all.

```
</xs:documentation>
</xs:annotation>
<xs:attribute name="Channel" type="xs:string" use="required">
  <xs:annotation>
    <xs:documentation>
      Name of the channel we will send events to.
      For example, "SCHEDULING_CHANNEL".
```



```

        </xs:documentation>
    </xs:annotation>
</xs:attribute>
<xs:attribute name="ID" type="xs:string" use="required">
    <xs:annotation>
        <xs:documentation>
            The IDL id of the ALMA event we will send an event to.
            For example, "IDL:alma/FRIDGE/temperatureDataEvent:1.0".

            If the event element contains some text, the simulator framework
            will not create a random instance of this event but will instead evaluate
            the element's text to produce the event.
        </xs:documentation>
    </xs:annotation>
</xs:attribute>
<xs:attribute name="Frequency" use="optional" default="0">
    <xs:annotation>
        <xs:documentation>
            The floating point number of seconds the simulator framework should wait
            before sending an event after the previous event. The default value of 0 implies
            the framework should only send one event and then stop.
        </xs:documentation>
    </xs:annotation>
    <xs:simpleType>
        <xs:restriction base="xs:float">
            <xs:minInclusive value="0"/>
        </xs:restriction>
    </xs:simpleType>
</xs:attribute>
</xs:complexType>
<!--#####-->
<xs:complexType name="eventResponse" mixed="true">
    <xs:annotation>
        <xs:documentation>
            The eventResponse type is an XML element describing ALMA events and channels.
            What this element does is tell the simulator framework that a given event type on a
            given channel name should be subscribed to and when an event of the correct type
            is received:
            1. A block of Python code existing within this element should be executed and/or
            2. Another event should be sent out as a response.
        </xs:documentation>
    </xs:annotation>
    <xs:attribute name="IncomingChannel" type="xs:string" use="required">
        <xs:annotation>
            <xs:documentation>
                Name of the channel we will subscribe to.
                For example, "CONTROL_CHANNEL".
            </xs:documentation>
        </xs:annotation>
    </xs:attribute>
    <xs:attribute name="IncomingID" type="xs:string" use="required">
        <xs:annotation>
            <xs:documentation>
                The IDL id of the ALMA event we are subscribing to.
                For example, "IDL:alma/FRIDGE/temperatureDataEvent:1.0".
            </xs:documentation>
        </xs:annotation>
    </xs:attribute>

```

```

    </xs:annotation>
  </xs:attribute>
  <xs:attribute name="OutgoingChannel" type="xs:string" use="optional" default="">
    <xs:annotation>
      <xs:documentation>
        Name of the channel we will send an event to as a response to an incoming event.
        For example, "SCHEDULING_CHANNEL".
        This attribute is not used unless the OutgoingEventId attribute is modified from the
        default value.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>
  <xs:attribute name="OutgoingID" type="xs:string" use="optional" default="">
    <xs:annotation>
      <xs:documentation>
        The IDL id of the ALMA event we will send out as a response to an incoming event.
        For example, "IDL:alma/FRIDGE/temperatureDataEvent:1.0".

        If the OutgoingChannel attribute is not modified from the default value, the event will
        be sent to the IncomingChannel.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>
  <xs:attribute name="Delay" type="xs:double" use="optional" default="1">
    <xs:annotation>
      <xs:documentation>
        The floating point number of seconds the simulator framework should wait
        before sending an event in response to receiving an event of IncomingEventId
        type. This attribute is ignored if OutgoingEventId has not been changed from its default
        value.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>
  <xs:attribute name="MissedEventChance" use="optional" default="0">
    <xs:simpleType>
      <xs:restriction base="xs:double">
        <xs:minInclusive value="0"/>
        <xs:maxExclusive value="1"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
</xs:complexType>
<!-- #####-->
<xs:complexType name="corbaAttribute" mixed="true">
  <xs:annotation>
    <xs:documentation>
      Type which defines CORBA attributes such as BACI properties.
      This XML element should contain a block of Python code with the last line being the return value.
      It could be something similar to:
      return "this string value for the following IDL - readonly attribute string stuff;"
    </xs:documentation>
  </xs:annotation>
  <xs:attribute name="Name" type="xs:string" use="required">
    <xs:annotation>
      <xs:documentation>
        Name of the CORBA attribute.
      </xs:documentation>
    </xs:annotation>
  </xs:attribute>

```

```

        </xs:documentation>
    </xs:annotation>
</xs:attribute>
<xs:attribute name="Timeout" type="xs:float" use="optional" default="1">
    <xs:annotation>
        <xs:documentation>
            Amount of time in floating point seconds that must pass before the simulator framework returns
            control to the caller.
        </xs:documentation>
    </xs:annotation>
</xs:attribute>
</xs:complexType>
<!--#####-->
<xs:complexType name="lifeCycleMethod" mixed="true"/>
<!--#####-->
<xs:complexType name="corbaMethod" mixed="true">
    <xs:annotation>
        <xs:documentation>
            Type which defines a CORBA component method.
            This XML element should contain a block of Python code with the last line being the return value.
            If the CORBA method is void, the final line should return None:
print "beginning"
while 1:
    #do some stuff

None
        </xs:documentation>
    </xs:annotation>
<xs:attribute name="Name" type="xs:string" use="required">
    <xs:annotation>
        <xs:documentation>
            Name of the CORBA method. For example, "on".
        </xs:documentation>
    </xs:annotation>
</xs:attribute>
<xs:attribute name="Timeout" type="xs:double" use="optional" default="5">
    <xs:annotation>
        <xs:documentation>
            Amount of time in floating point seconds that must pass before the simulator framework returns
            control to the caller.
        </xs:documentation>
    </xs:annotation>
</xs:attribute>
</xs:complexType>
<!--#####-->
<xs:element name="SimulatedComponent">
    <xs:annotation>
        <xs:documentation>
            CDB XMLs located in the $ACS_CDB/alma/simulated/* section of the CDB must validate against this
schema.
            SimulatedComponent defines the behavior of components using the generic IDL simulator framework.
        </xs:documentation>
    </xs:annotation>
</xs:complexType>
    <xs:sequence>
        <xs:element name="pythonImports" type="imports" minOccurs="0"/>

```

```

<xs:element name="initialize" type="lifeCycleMethod" minOccurs="0"/>
<xs:element name="cleanUp" type="lifeCycleMethod" minOccurs="0"/>
<xs:element name="_corbaMethod" type="corbaMethod" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="_corbaAttribute" type="corbaAttribute" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="_almaEvent" type="event" minOccurs="0" maxOccurs="unbounded"/>
<xs:element name="_almaEventResponse" minOccurs="0" maxOccurs="unbounded">
  <xs:complexType mixed="true">
    <xs:complexContent mixed="true">
      <xs:extension base="eventResponse"/>
    </xs:complexContent>
  </xs:complexType>
</xs:element>
</xs:sequence>
<xs:attribute name="AllowInheritance" type="xs:boolean" use="optional" default="true">
  <xs:annotation>
    <xs:documentation>
      This attribute is used by the simulator framework to determine if it's OK to look at superclasses
      of the component also residing within the $ACS_CDB/alma/simulated/* section of the CDB.
      Change it to 'false' and the simulator will only use the current XML.
    </xs:documentation>
  </xs:annotation>
</xs:attribute>
</xs:complexType>
</xs:element>
<!-- #####-->
</xs:schema>

```

### 3.5 Graphical User Interface

To be redone for ACS 6.0!

## 4 Usage

### Getting Started

Regardless of how you intend on using the simulator, there are a few steps that must be performed. Obviously an IDL file must be created which defines an IDL interface derived from *ACS::ACSCComponent* or *ACS::CharacteristicComponent*. There are other ACS documents and examples which describe this process in detail so we will assume you have already created an IDL file, compiled it, and installed it into an INTROOT area.

### Editing the CDB to Specify Simulated Components

Just like all other components in ALMA software, your simulated component will need at least one entry in the ACS Configuration Database. Specifically, *\$ACS\_CDB/CDB/MACI/Components/Components.xml* must be modified to tell the Python

container to activate the component using a special Python package developed by ACS. For example, the entry for the “HELLODEMO1” component found in the default ACS CDB installed with ACS would be changed from:

```
<_ Name="TEST_RPS_1"
  Code="acsexmplRampedPowerSupplyImpl"
  Type="IDL:alma/RampedPS/RampedPowerSupply:1.0"
  Container="cppContainer"/>
```

to:

```
<_ Name="TEST_RPS_1"
  Code="Acssim.Servants.Simulator"
  Type="IDL:alma/RampedPS/RampedPowerSupply:1.0"
  Container="pyContainer"/>
```

The *Code* attribute of the component must appear verbatim as depicted above, but you are completely free to choose a different reasonable name for the *Container* attribute so long as it is started as a Python container.

## Editing the CDB to Specify Default Behavior for a Simulated Component

After completing the previous step, you’ve actually done everything that is required to instantiate and simulate the methods and attributes of your component. That is, the Python package described by the *Code* attribute for your component in *\$ACS\_CDB/CDB/MACI/Components/Components.xml* will use the CORBA Interface Repository to automatically generate random return values on the fly!

In some cases completely random values are desired, but in other cases a developer might want to specify that “abc” method always returns the value “x”. To deal with this, the ACS component simulator framework allows the developer to optionally place an XML file (which conforms to the *SimulatedComponent.xsd* schema found in *\$ACSROOT/config/CDB/schemas*) with a name identical to that of the component in *\$ACS\_CDB/CDB/alma/simulated/*. In this XML the developer can specify the return values or exceptions to be thrown on any given method invocation or attribute access. It’s important to note that the developer can omit method and attribute definitions entirely from this XML without worrying about whether or not they will fail (e.g., the framework will automatically implement them if they’re not found in the XML). The rest of this section is dedicated to an example which depicts how to implement a simple simulated component XML configuration file.

Let’s take a look at an example simulated component XML configuration file based on the *RampedPowerSupply* IDL interface found in the *acsexmpl* CVS module:

```
1. <SimulatedComponent xmlns="urn:schemas-cosylab-
   com:SimulatedComponent:1.0"
```

```
2.         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
3.         AllowInheritance="true">
4.
5.     <pythonImports>
6. import acstime
7.     </pythonImports>
8.
9.     <initialize>
10. #the lifecycle and component methods defined within IDL
11. #are not actual instance methods of the simulated component.
12. #instead, they are actually proxy functions which get indirectly
13. #called by the simulated component. because of this, the
14. #simulator framework provides a reference to the raw component
15. #as the final param within the parameters list. using this
16. #gives us access to the container services!!!
17. logger = parameters[0].getLogger()
18. logger.logInfo("logging directly from XML using a nice trick!!!")
19.
20. #create both of the objects
21. compl = ACSErr.Completion(0L, 0L, 0L, ())
22. cbdo = ACS.CBDescOut(0L, 0L)
23.
24. #now use some special API functions defined within Acssim.Goodies
25. #to store these local objects globally
26. setGlobalData("compl", compl)
27. setGlobalData("cbdo", cbdo)
28.
29. #get a reference to the power supply component
30. ps = parameters[0].getComponent("TEST_PS_1")
31.
32. #get a reference to the readback property
33. readback_prop = ps._get_readback()
34.
35. #attach a new member to the simulated component to
36. #provide easy access to the readback property
37. parameters[0].readback_prop = readback_prop
38.     </initialize>
39.
40.     <cleanUp>
41. #be sure to cleanUp everything!
42. parameters[0].releaseComponent("TEST_PS_1")
43.     </cleanUp>
44.
45. <_corbaMethod Name="stopRamping" Timeout="0">
```

```

46. print "Simulated stopRamping (alma/simulated/TEST_RPS_1 CDB section):"
47. print "  messing with another component and invoking the callback's done
    method."
48. print
49.
50. #get the readback property's value
51. readback_prop = parameters[len(parameters)-1].readback_prop
52. readback_value = readback_prop.get_sync()[0]
53.
54. print "Value of TEST_PS_1's readback property is:", readback_value
55. parameters[0].done(getGlobalData("compl"),
56.                    getGlobalData("cbdo"))
57. print
58. return
59.   </_corbaMethod>
60.
61.   <_almaEventResponse IncomingChannel="fridge"
62.     IncomingID="IDL:alma/FRIDGE/temperatureDataBlockEvent:1.0
    "
63.     OutgoingChannel="TIME_CHANNEL"
64.     OutgoingID="IDL:alma/acstime/Duration:1.0"
65.     Delay="0.123"
66.     MissedEventChance="0.43">
67. print "Simulated almaEventResponse(temperatureDataBlockEvent):", parameters
68.
69. #create a duration to be sent as an event. it doesn't matter
70. #that this isn't technically an event - all that matters is
71. #that Duration is an IDL struct
72. my_event = acstime.Duration(345L)
73. return my_event
74.   </_almaEventResponse>
75.
76. </SimulatedComponent>

```

**This file would be placed in \$ACS\_CDB/CDB/alma/simulated/TEST\_RPS\_1/ and named 'TEST\_RPS\_1.xml' for a component named 'TEST\_RPS\_1'.**

- 1 This XML file provides the partial implementation of the TEST\_RPS\_1 component. Because of this XML's location within the ACS CDB (i.e., \$ACS\_CDB/CDB/alma/simulated/), this simulated behavior description is tied into the TEST\_RPS\_1 component and in no way affects the behavior of other simulated components implementing the IDL:alma/RampedPS/RampedPowerSupply:1.0 interface.

There are several things to take note of here:

1. Everything defined within XML elements consists of Python code. You must obey the whitespace rules of Python within these.
2. Not all methods and BACI properties defined in IDL for this component are defined here. This is unnecessary as the simulator framework will dynamically generate methods and properties that have not been defined by some means available to the end-user.
3. Implementations of inherited methods defined in other parts of the CDB will be used where applicable. In this specific case, see `$ACS_CDB/CDB/alma/simulated/interfaces/alma/PS/PowerSupply/1.0` and also `$ACS_CDB/CDB/alma/simulated/interfaces/alma/RampedPS/RampedPowerSupply/1.0`. These locations found within `$ACS_CDB/CDB/alma/simulated/interfaces` can be determined by the CORBA interface repository ID of your component (i.e., "IDL:alma/PS/PowerSupply:1.0").
4. With respect to 3, one should also note that it's perfectly acceptable to override methods defined within `$ACS_CDB/CDB/alma/simulated/alma/*`. All simulated components should be derived from the `SimulatedComponent` XML element. **Pay close attention to the fact we set the `AllowInheritance` attribute to true. This means we want the simulator to search through other sections of the `$ACS_CDB` for (inclusive) supers of the `RampedPowerSupply` IDL interface which also specify simulated behavior.**
- 5 Section where normal Python imports are performed. By importing modules/packages here, they will be accessible throughout the rest of the XML.
- 9 Here we override a lifecycle method. This is being done so that we only have to create a `Completion` and `Callback` descriptor a single time rather than with each invocation of the `startRamping` method to invoke the "done" method of the `Callback` param.
- 40 Another lifecycle method.
- 45 A CORBA method is implemented. This particular method, `stopRamping`, accepts as its first parameter a `CallbackVoid` object. We in turn play around with the callback before returning.

## Important Notes:



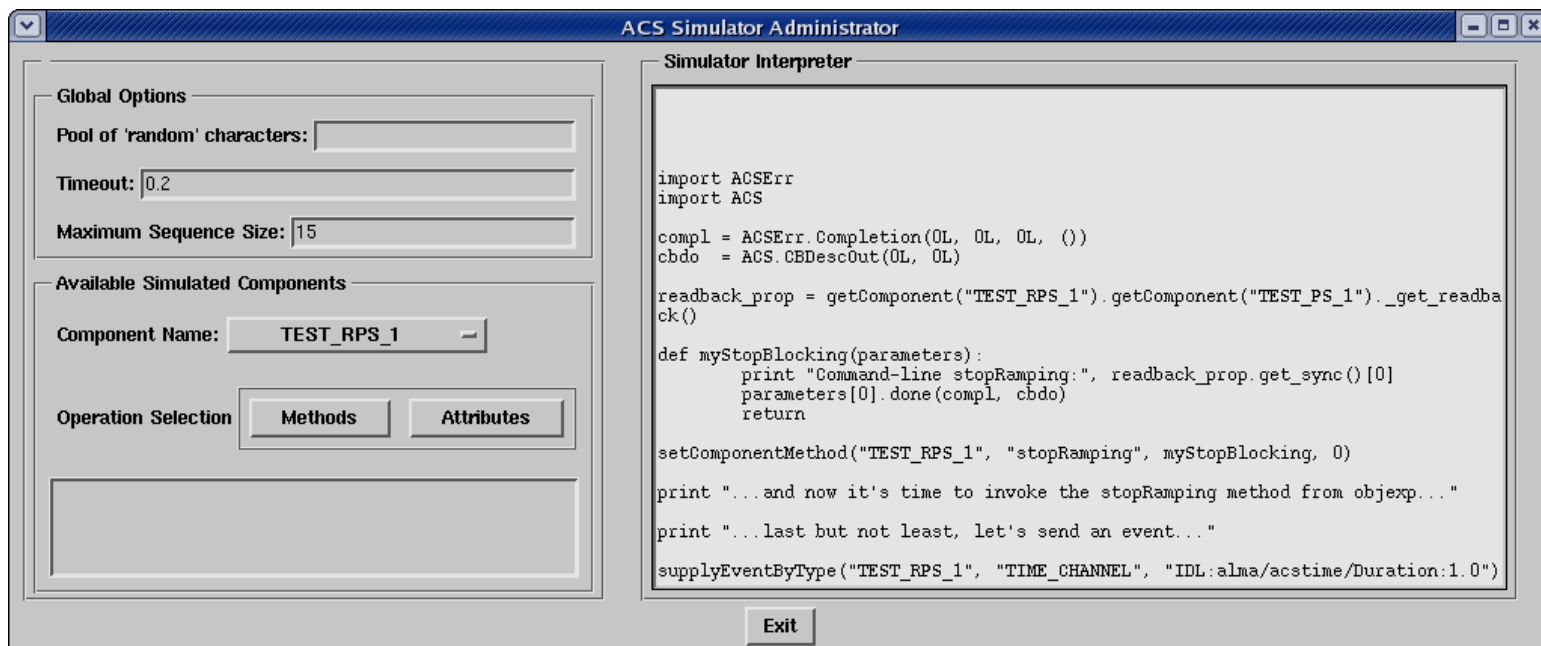
- CORBA attributes that are specified in the CDB and not BACI properties should have `_get_` prepended to the *Name* attribute of the *\_attribute* XML element.
- **The example presented above is a subset of a complete example found in the `acspyexmpl` module.** The rest of the example can be found in `acspyexmpl/test/CDB/alma/simulated/interfaces/alma/PS/PowerSupply/1.0/` and `acspyexmpl/test/CDB/alma/simulated/interfaces/alma/RampedPS/RampedPowerSupply/1.0/`

## Using Interactive Container Sessions

At times it can be quite useful to change the behavior of a simulated method or attribute at run-time. With recent additions to the ACS Python framework this is now possible! Upon starting the first simulated component, a widget now pops up which allows you to enter in Python commands as if they were being inputted into an interactive Python sessions. This in itself is quite powerful, but without an API to configure the run-time behavior of simulated components it is somewhat lacking. **An API has been created in the form of the `Accsim.Goodies Python` module.** Developers interested in using the API should look at the pydoc for the appropriate methods which are generally in the format `setXXX` where `XXX` is some configurable data. Some of the configurable items are:

- A standard timeout for all methods and attributes that are dynamically implemented by the framework
- The maximum sequence size for CORBA sequences
- Associating a new timeout, function to be executed, etc for a particular component's method or attribute

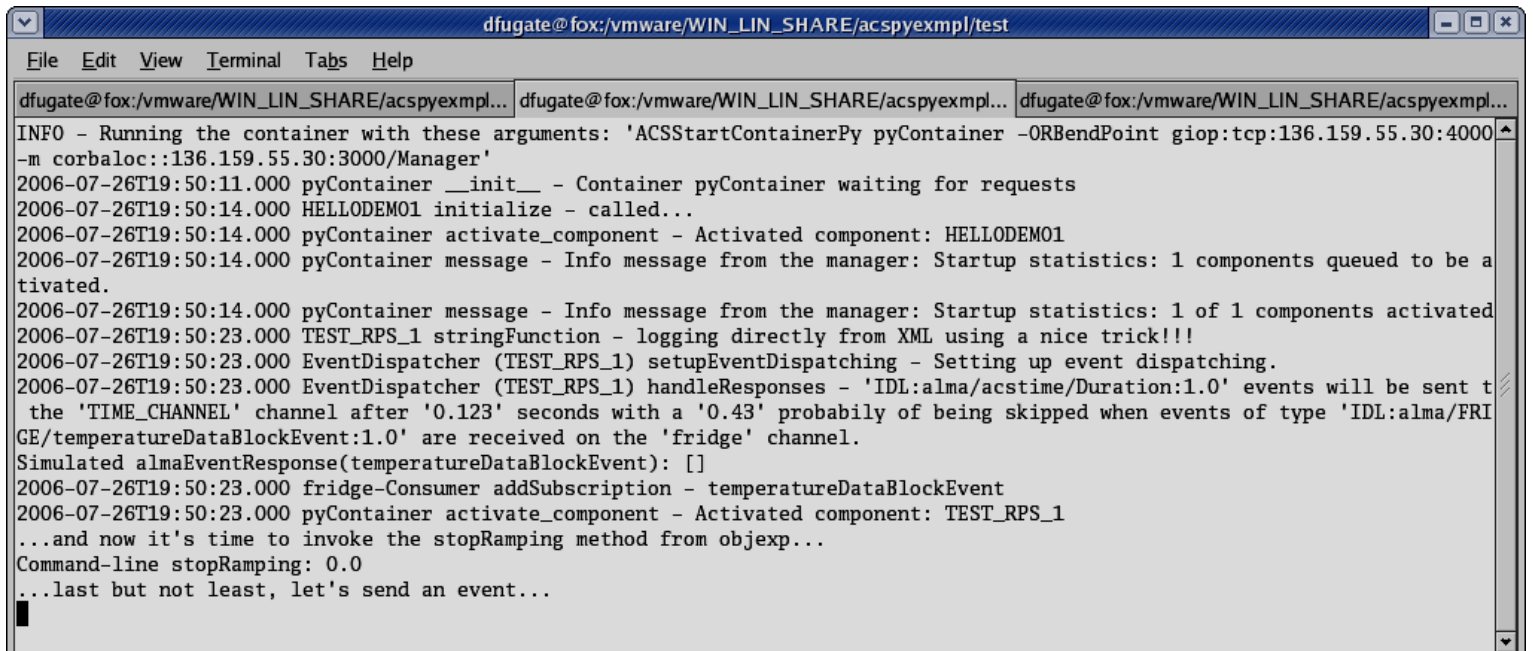
The following is a sample interactive session in which the behavior of `TEST_RPS_1`'s `stopRamping` method is implemented in a similar fashion to the previous CDB example:



## Special Notes:

- After defining functions, make sure you add one extra newline after the last line of your function.
- When defining functions to simulate component methods using the “setComponentMethod” API function; your function should only accept one parameter regardless of how many in/inout/out parameters the IDL method defines. This parameter corresponds to a list of the real parameters passed to the IDL component. To further illustrate this, see the *stopRamping* implementation above.
- After striking the “Enter” key, all characters on the current line are sent to the real Python interpreter to be evaluated.
- While you may be able to “Backspace” your way up to the previous line, this has no real effect.
- Messages from the real Python interpreter sent to stderr will end up in red text within the widget. Non-erroneous messages are sent to the real Python console as would occur normally.

The text entered into the Simulator Interpreter results in the following output from the real console:



```

dfugate@fox:/vmware/WIN_LIN_SHARE/acspyexmpl/test
File Edit View Terminal Tabs Help
dfugate@fox:/vmware/WIN_LIN_SHARE/acspyexmpl... dfugate@fox:/vmware/WIN_LIN_SHARE/acspyexmpl... dfugate@fox:/vmware/WIN_LIN_SHARE/acspyexmpl...
INFO - Running the container with these arguments: 'ACSStartContainerPy pyContainer -ORBEndPoint giop:tcp:136.159.55.30:4000
-m corbaloc::136.159.55.30:3000/Manager'
2006-07-26T19:50:11.000 pyContainer __init__ - Container pyContainer waiting for requests
2006-07-26T19:50:14.000 HELLODEM01 initialize - called...
2006-07-26T19:50:14.000 pyContainer activate_component - Activated component: HELLODEM01
2006-07-26T19:50:14.000 pyContainer message - Info message from the manager: Startup statistics: 1 components queued to be a
tivated.
2006-07-26T19:50:14.000 pyContainer message - Info message from the manager: Startup statistics: 1 of 1 components activated
2006-07-26T19:50:23.000 TEST_RPS_1 stringFunction - logging directly from XML using a nice trick!!!
2006-07-26T19:50:23.000 EventDispatcher (TEST_RPS_1) setupEventDispatching - Setting up event dispatching.
2006-07-26T19:50:23.000 EventDispatcher (TEST_RPS_1) handleResponses - 'IDL:alma/acstime/Duration:1.0' events will be sent t
the 'TIME_CHANNEL' channel after '0.123' seconds with a '0.43' probability of being skipped when events of type 'IDL:alma/FRI
GE/temperatureDataBlockEvent:1.0' are received on the 'fridge' channel.
Simulated almaEventResponse(temperatureDataBlockEvent): []
2006-07-26T19:50:23.000 fridge-Consumer addSubscription - temperatureDataBlockEvent
2006-07-26T19:50:23.000 pyContainer activate_component - Activated component: TEST_RPS_1
...and now it's time to invoke the stopRamping method from objexp...
Command-line stopRamping: 0.0
...last but not least, let's send an event...

```

## Using the GUI

The first time a simulated component is activated, a graphical user interface is automatically started which allows configuring the run-time behavior of simulated components in the same fashion as the API described previously. Once this GUI has been closed it cannot be restarted without restarting the Python container so be very careful about this.

Figure 1 – the main panel

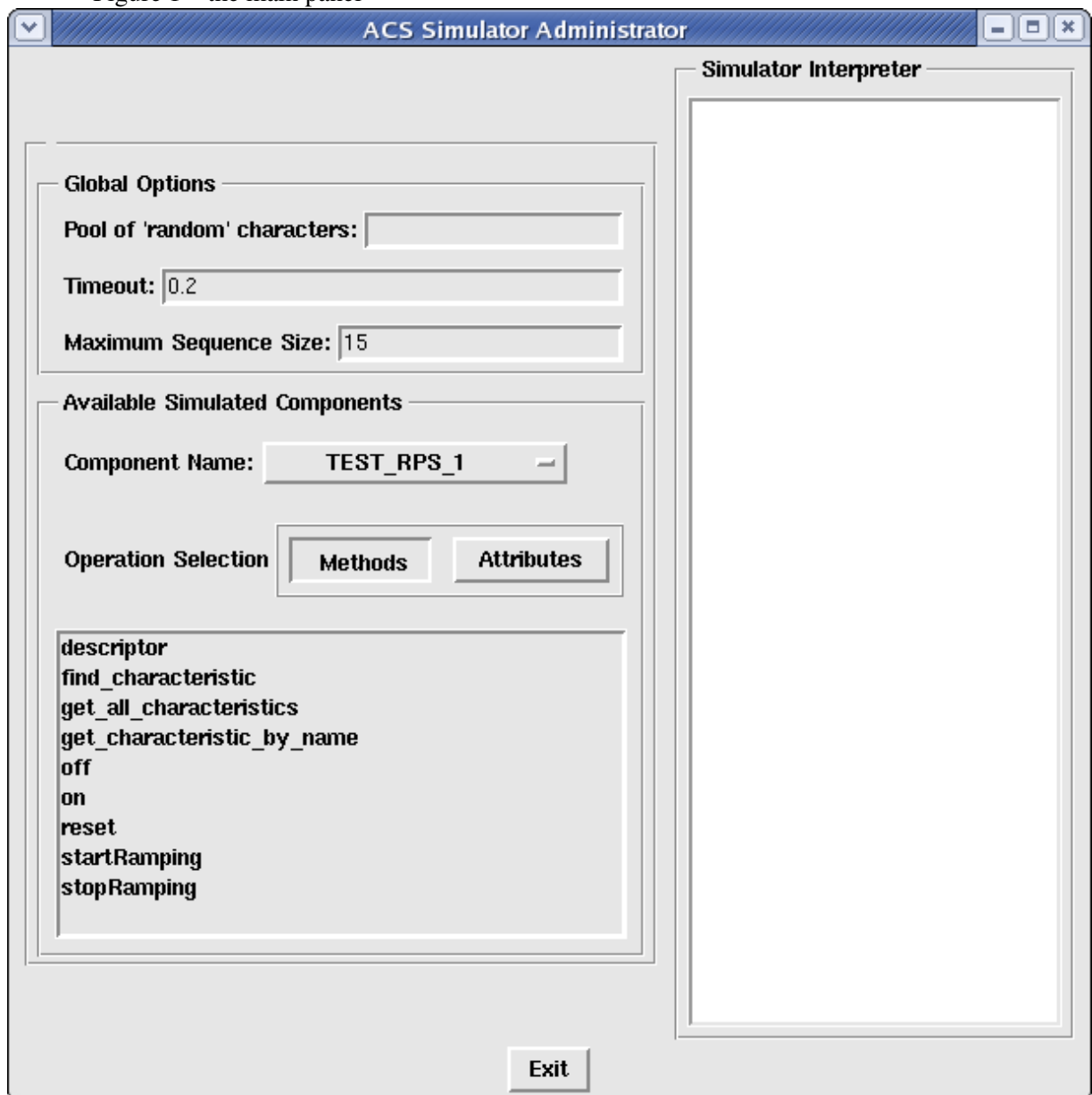
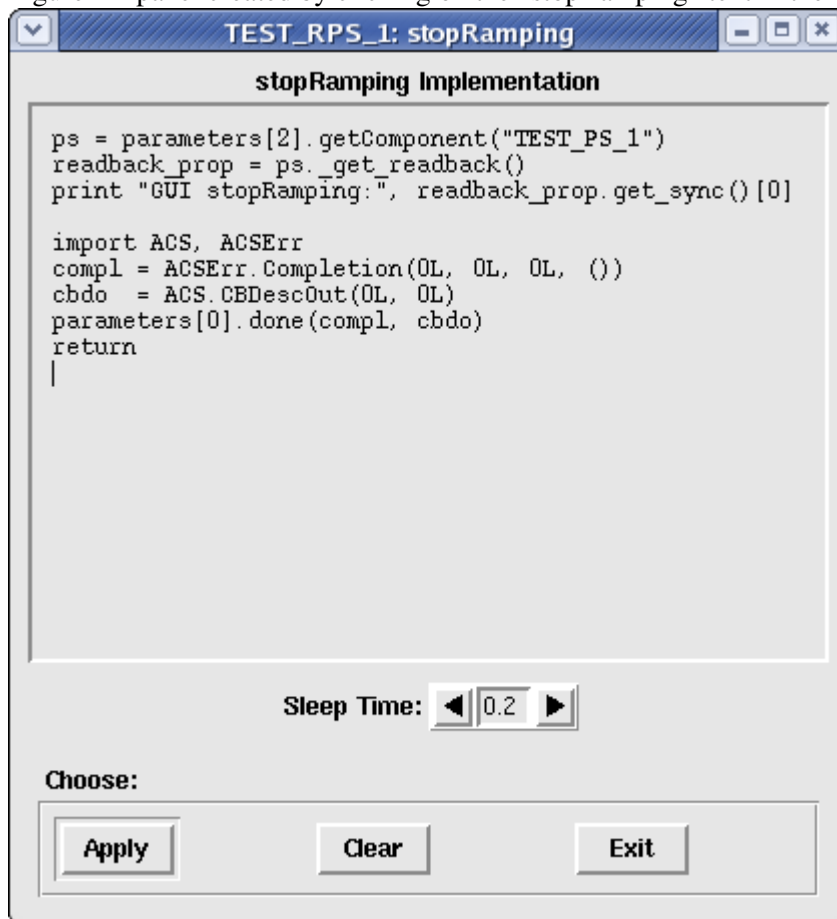


Figure 2 – panel created by clicking on the “stopRamping” text in the main GUI



**Then once you’ve finished implementing the method, simply click the “Apply” button!**

## Special Notes

### Accessing Method Parameters

The *in* and *inout* parameters passed to simulated methods are accessible from the CDB, API, and GUI. They can be accessed via the *parameters* variable (a Python list) which is automatically placed in the proper namespace for the developers to use in the case of the CDB and GUI. As for the API, faux method implementations should always accept only a single parameter – parameters.

The ordering of parameters within this list is identical to that of the IDL->Python mapping. For example, say we are given an IDL method defined like this:

```
string methodA(in boolean x, inout long y, out double z)
```

In this case, *parameters* could be equal to `[0, 34L, comp_ref]` where `0` maps to the *x* boolean parameter and `34L` maps to the *y* parameter, and `comp_ref` is a literal reference to the raw Python component being simulated (meaning you have direct access to it's ContainerService methods). Under the IDL->Python mapping, *z* is omitted from the list of parameters.

## 5 Known Problems and Issues

None at the moment.

...

## 6 Appendix

### RampedPowerSupply IDL Interface

### PowerSupply IDL Interface