OpenCPI Application Development Reference

(ADR)

Revision History

Revision	Description of Change	Date
1.01	Creation, in part from previous Application Control Interface document	2012-12-10
1.1	Add package naming and ocpirun flags that apply to all, not one, instance. Also the DumpFile attribute. Also some clarifications about property value formats (struct etc.).	2013-02-11
1.2	Add container ordinal and platform options for ocpirun, clarify array value format. Add a few missing Worker methods.	2013-02-28

Table of Contents

1 Introduction

1.1 References

This document depends on several others.

Table 1 - Table of Reference Documents

Title	Published By	Link
OpenCPI Technical Summary	OpenCPI	Public URL: http://opencpi.org/documentation.php
OpenCPI Component Development Guide	OpenCPI	Public URL: http://opencpi.org/documentation.php

1.2 Purpose

The purpose of this document is to specify the ways that applications can be created and executed in OpenCPI. One way is XML-based and the other uses a C++ interface for launching and controlling OpenCPI applications. Having one or more libraries of prebuilt, ready-to-run component implementations (workers) is a prerequisite for running applications. How such component implementations are developed is defined in the ***Component Development Guide***. Creating and running applications for OpenCPI does *not* require the knowledge of how components are developed, but it does require some knowledge of how they are *specified*. Component specifications are discussed briefly in this document, and described in full in the ***Authoring Model Reference*** document.

OpenCPI applications are defined as assemblies of component instances with connections among them. They can be specified three different ways:

1. A standalone XML document (text file).
2. An XML document embedded in, and manipulated by, a C++ program.
3. A set of C++ API calls using the Application Control Interface API (ACI).

This document thus specifies:

- *The format and contents of the XML documents*
- *A utility program that directly executes the applications defined in XML*
- *A C++ API for manipulating and executing XML-based applications*
- *A C++ API for dynamically creating and executing component-based applications*

1.3 Overview

OpenCPI uses several terms when describing component-based applications. In particular:

Component: a specific function with which to compose applications. Components are described by an XML document called a *component specification*.

Instance: the use of a component in an application (a part of an assembly).

Assembly: the composition of instances that define an application

Worker: a concrete implementation of a component, in three contexts: source code, compiled code for some target platform, runtime object executing the compiled code.

Container: the OpenCPI execution environment on some platform that will execute workers (i.e. where they execute).

Port: a communication interface of a component or worker, with which they communicate with other components/workers.

Property: a configuration value applied to a worker to control its function. Components have defined properties with defined data types, and workers implement those properties. Workers (implementations) may have additional properties beyond those defined by the component being implemented.

Platform: a particular type of processing hardware and software that can host a container for executing workers.

Artifact: a file containing binary code for one or more workers, built for a platform.

Library: a collection of artifacts in a hierarchical file system directory structure.

Package: a name scope for *component specifications* (*not* implementations).

The OpenCPI execution framework for component-based applications is based on ***workers*** executing in ***containers*** (on ***platforms***), communicating via their ***ports***, and configured via their ***properties***. The ***workers*** are runtime instances of ***component*** implementations realized in ***artifact*** files. The term "component" represents the abstract functionality. The term "*artifact*" is used as a technology neutral term which represents a compiled binary file that is the resulting *artifact* of compiling (or for FPGAs, "synthesizing" etc.) and linking some source code that implements some ***components***. In fact, we use the term "***worker***" both for a specific (coded) implementation of a ***component***, as well as the runtime instances of that implementation.

The build process results in ***artifacts*** that can be loaded as needed and used to instantiate the runtime ***workers***. Typical ***artifacts*** are "shared object" or "dynamic library" files on UNIX systems for software workers, and "bitstreams" for FPGA ***workers***. While it is typical for ***artifacts*** to hold the implementation code for one ***worker***, it is also common to build artifact files that contain multiple ***worker*** implementations (hence the term "shared library").

OpenCPI ***applications*** are created by specifying which ***components*** should be instantiated (using some implementation), how the resulting ***workers*** should be connected, and how they should be configured via their properties. Specifying an instance is based on the name of the *component specification*. The runtime software uses this name to search the available component libraries for available implementations in artifacts, and matches those (binary) implementations to the available containers (processors of various types), running on platforms. The result of the search is a set of potential candidate workers for each instance. To be a candidate, an implementation must be able to execute in some available container.

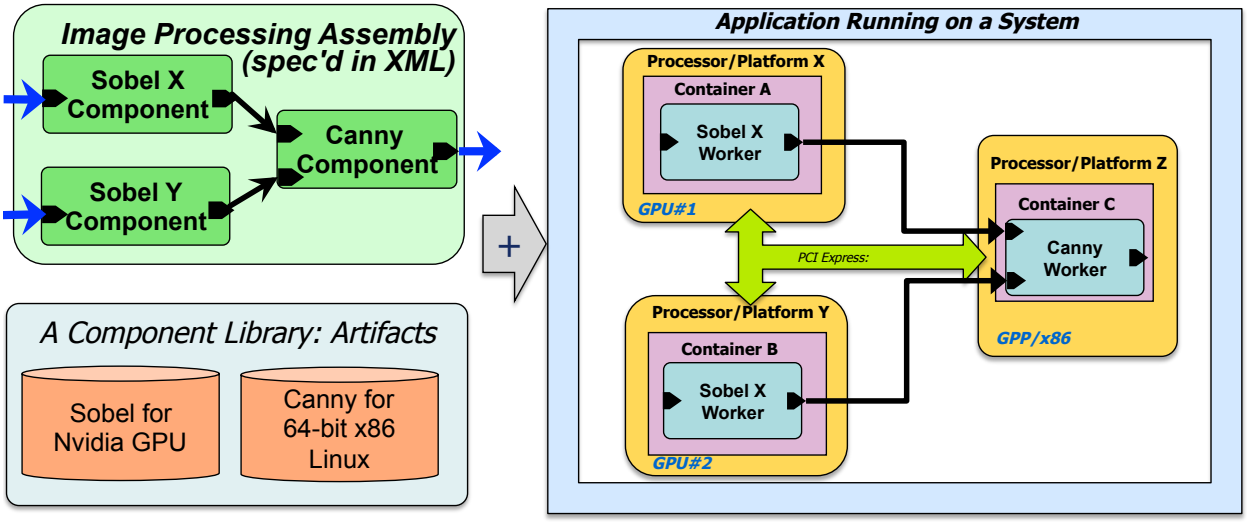
The name of a component specification can be prefixed with a ***package*** name (followed by a period). This allows components to be specified by different organizations, while

still allowing any implementation found in a library to satisfy any organization's component specifications. E.g., my project can have an additional, alternative implementation of a component specified in another library, or can define its own specification for a component with the same name.

Finally, to actually run the application, the "deployment decision" is made for each instance: *which implementation/artifact* should be used, and *which container* (on some platform) should it run in.

Unless a specific implementation is indicated, the OCPI_LIBRARY_PATH environment variable is used to indicate a list of colon-separated directories, which are searched for artifact files containing component implementations. The directories are searched recursively.

The relationships between applications, artifacts, containers, workers, platforms etc. is shown in the following diagram:



An Application of Components Deployed on a System

2 OpenCPI Application Specification (OAS) XML Documents

This section defines the XML document format for describing OpenCPI applications. Such XML documents may be held in files or in text strings within a program. They describe an assembly (collection) of components, along with their interconnections and configuration properties. An OAS may be directly executed using the "ocpirun" utility program described below. An OAS may also be constructed and/or manipulated programmatically and dynamically, and executed using an API.

The primary contents of the OAS are *component instances*. When the author of an OAS specifies a component *instance*, they are referring to a component *specification*. They are saying: I need a component implementation that meets this *specification*. Normally, the OAS says only that, and does *not* say which particular implementation of that component spec (i.e. which worker) should be used. This allows the OAS to be used in a variety of different configurations of hardware and different libraries of component implementations.

A very simple example of an OAS is below, showing an application that reads data from a file, adds 1 to each data value, and writes the result.

```
<application>
  <instance component='file_read' connect='add1'>
    <property name='filename' value='test.input'/>
  </instance>
  <instance component='add1' connect='file_write'/>
  <instance component='file_write'>
    <property name='filename' value='test.outputwrong'/>
  </instance>
</application>
```

Each instance specifies the component, some properties, and a connection.

2.1 Quick XML Introduction

XML documents are text files that contain information formatted according to XML (E**X**tensible **M**arkup **L**anguage) syntax and structured according to a particular application-specific "XML schema". The textual XML information itself is formatted into ***elements***, ***attributes***, and ***textual content***. The OAS XML Schema does not use or allow ***textual content*** at this time. XML ***elements*** have ***attributes*** and ***child elements*** (forming a hierarchy of elements). XML ***elements*** take two forms. The simpler one is when an element has no child (embedded) elements and no ***textual content***. It looks like this (for element of type , "xyz", with attribute "abc"):

```
<xyz abc='123'/>
```

Thus the element begins with the "<" character and the element type, and is terminated with the "/>" characters. Attributes have values in single or double-quotes. Any white space, indentation, or new lines can be inserted for readability between the element name and attributes or between attributes. Thus the above example could also be:

```
<xyz
  abc="123"
  />
```

When the element has child elements (in this case a child element of type “ccc” with attribute “cat”), it looks like:

```
<xyz abc=”123”>
  <ccc cat=”345”/>
</xyz>
```

In this case the start of the “xyz” element (and its attributes), is surrounded by <>, and the end of the “xyz” element is indicated by “</xyz>. So, an XML Schema defines which Elements, Attributes, and child Elements the document may contain. Every XML document has a single top-level element that must be structured (attributes and sub-elements) according to the XML schema.

An element can be entered directly (as above) or entered by referring to a separate file that contains that element. So the example above might have a file “ccc1.xml” containing:

```
<ccc cat=”345”/>
```

And then a top-level file called “xyz1.xml” containing:

```
<xyz abc=”123”>
  <xi:include href=”ccc1.xml”/>
</xyz>
```

However, the schema specifies which elements are allowed to be top-level elements in any file. All element and attribute names used in OpenCPI are case ***IN***sensitive.

There are various built-in attributes that are part of the XML system itself and elements (such as the xi:include element above), and are required in certain cases (when processed outside OpenCPI). In particular, the top-level element of a top-level file (not an included file) must sometimes have these built-in, or “boilerplate” attributes:

To (optionally) specify the schema according to which this document is structured, these attributes would be included in the top-level element of a file:

```
xmlns:x="http://www.w3.org/2001/XMLSchema-instance"
x:schemaLocation="http://www.opencpi.org/xml schema.xsd"
xmlns="http://www.opencpi.org/xml"
```

To allow the use of the “xi:include” feature above, the top-level element in a file with such inclusions inside it may need to contain the attribute:

```
xmlns:xi="http://www.w3.org/2001/XInclude"
```

If all these boilerplate attributes were included in the example above, it would look like:

```
<xyz
      xmlns:x="http://www.w3.org/2001/XMLSchema-instance"
      x:schemaLocation="http://www.opencpi.org/xml schema.xsd"
      xmlns="http://www.opencpi.org/xml"
      xmlns:xi="http://www.w3.org/2001/XInclude" abc=”123”>
  <xi:include href=”ccc1.xml”/>
</xyz>
```

The schema reference attributes allow the files to be processed with very generic tools, since the files contain the reference to the schema they are following. If they are not included, then they must be supplied as command-line arguments on the tools that

process these files. With these attributes included, any XML processor can process the files without being told about the schema.

OpenCPI itself never requires these boilerplate items, but some separate XML processing tools do have this requirement in some cases.

In OpenCPI all attributes are defined with specific data types and/or formats. When an attribute is defined as the "Boolean" type, the default value (used when the attribute is not specified) is "false" unless otherwise noted. All element and attribute names are *case INsensitive*.

2.2 Top Level Element in an OAS: application

The top-level element in every OAS document (file or string) is the "application" element. Application elements contain child elements that are either ***instance*** or ***connection***, and have attributes that are ***name***, ***done***, and ***maxprocessors***. These are described below.

2.2.1 Name attribute (optional)

The "name" attribute of an application is simply used in various error messages and other debug log messages. It has no functional purpose, only documentation and labeling.

2.2.2 Package attribute (optional)

The "package" attribute of an application is used as a default package prefix for all instances in the assembly. Any instance's component attribute that does not have a package prefix is assumed to be in the package indicated by this attribute. When not specified, the default package prefix for all components mentioned in the assembly is "local". The prefix of the core OpenCPI component library is "ocpi". If you are using mostly components in that library, you might include " package='ocpi' " as an attribute. If you are using only components specified in your own library of components (which has a default prefix of "local"), you could ignore this attribute and use no prefixes at all. See the "component" attribute of the "instance" element below.

2.2.3 Done attribute (optional)

The "done" attribute identifies the instance within the assembly that is used to determine when the application is "done" executing. When the indicated instance is "done", then the whole application is considered "done". If this attribute is not supplied, the application is considered "done" when *all* its instances are "done". This value of this attribute must match the "name" attribute of one of the instance elements, described below.

For some applications, and some components, there is no definition or functionality of being "done". In this case whatever mechanism started the application must decide when to stop it and shut it down.

2.2.4 MaxProcessors attribute (optional)

The "MaxProcessors" attribute indicates the maximum number of processors that should be used to run the application (thus all its instances). When instances are

allocated to run on processors, there is an algorithm that decides which processor (container) will run each instance. If this attribute is not set, the default behavior is to spread the instances across available processors, and use a "round-robin" policy when there are more instances than processors.

If this numeric parameter is set, it limits the number of processors used, if possible. If more are necessary to host the necessary workers, more will indeed be used in any case. An example of when this attribute is not effective is when the availability of implementations of each instance dictate that more processors are needed, such as when the only implementation available for an instance is for a particular processor, which must be used.

2.3 Instance Elements within the Application Element

The "instance" element is used as a child of the "application" element to specify a component instance in the application. It may have ***property*** child elements, and may have ***component***, ***name***, ***selection*** and ***connect*** attributes.

2.3.1 Component attribute (required)

The "component" attribute of an instance specifies the name of the component being instantiated. The value is a string used to find implementations for this instance in the available component libraries. It is the name assigned by the component developer to the component *specification* used as the basis for implementations. Component specifications are themselves XML documents/elements called OCS (OpenCPI Component Specification). They have names (used to match this attribute's value), and describe the ports and properties that apply to all implementations of that component.

This attribute is required, and answers the question: what function should this instance perform?

The process by which OpenCPI searches in libraries for implementations based on this attribute is described above.

This attribute may have a package prefix (ending in a period) to indicate which package contains the component specification indicated. If there is no prefix, the package prefix is taken from the default for the whole assembly, which is specified using the "package" attribute of the top-level "application" element.

2.3.2 Name attribute (optional)

The "name" attribute of an instance is optional, and provides a unique identifier for the instance within the application. If it is not supplied, one is assigned to the instance. If there is only one instance in the application for a component (i.e. the component is used only once), the assigned instance name (if none is supplied) is the same as the component name (without package prefix). If more than one such instance (of the same component) exists in the application, the assigned name is the component name (without package prefix) followed by the decimal ordinal of that instance among all those for the same component. Such ordinals are assigned starting with 0.

2.3.3 Selection attribute (optional)

This attribute optionally specifies how to choose among alternative implementations when more than one is available. This capability also provides a way for the application to specify minimum conditions on the candidate implementations found in the library.

The attribute value is an expression in the syntax of the C language, with all the normal operators. Logical expressions (e.g. "a == 1") return 1 on true and 0 when false. The variables that may appear in the expression are one of:

- Property names that have fixed (not runtime) values in the implementation
- Built-in identifiers that indicate well-known attributes of the implementation.

The built-in identifiers are:

- ***model***: the name of the authoring model of the implementation, e.g. "rcc".
- ***platform***: the name of the platform the implementation is built for, e.g. "x86_64" or "ml605".
- ***os***: the name of the operating system the implementation is built for, e.g. "linux".

The value of the selection expression is considered an unsigned number, where a higher number is better than a lower number, and zero is considered unacceptable. I.e. if the expression when evaluated for an implementation has a zero value, that implementation is not considered a candidate. A simple example might be:

```
model=="rcc"
```

This indicates that the model must be "rcc" since otherwise the expression's value will be zero. The example:

```
error_rate > 5 ? 2 : 1
```

indicates that the "error rate" property is relevant, and if greater than 5, it is better than when less than or equal to 5, but the latter is still acceptable.

If there is no selection expression, the "score" of the implementation is 1, unless it has hard-wired connections to other collocated workers (e.g. on an FPGA). In this case its value is 2.

An example of using the selection attribute is:

```
<application>
  <instance component="psd" selection='latency < 5'/>
  <instance component="demod" selection='model=="rcc"'/>
</application>
```

It indicates that the "psd" instance needs an implementation with latency less than 5, and the "demod" instance must have an implementation with an authoring model of "rcc".

2.3.4 Connect attribute (optional)

Connecting instances together in an assembly can be done one of two ways. Using the ***connect*** attribute of an instance is the simplest, but cannot express all connections. The "connection" child element of the "application" element can be used to express all types of connections. It is described later.

The ***connect*** attribute defines exactly one connection from an output port of this instance to an input port of another instance. Its value is the name of the other instance. If this instance only has one output port and the other instance only has only one input port, then these are implied. The optional ***from*** attribute specifies the name of the output port of this instance if needed (if there are more than one), and the optional ***to*** attribute specifies the name of the input port of the other instance (of there are more than one). An example using all three attributes is:

```
<application>
  <instance component="psd"
            connect='demod' from='myout' to='demod-in'/>
  <instance component="demod" selection='model=="rcc"'/>
</application>
```

This simple connection method is useful for the many components that have only one output port.

2.3.5 From attribute (optional)

This attribute is used to specify the name of the output port of this component instance in conjunction with using the "connect" attribute described above.

2.3.6 To attribute (optional)

This attribute is used to specify the name of the input port of the "other" component instance in conjunction with using the "connect" attribute described above.

2.3.7 External attribute (optional)

This attribute is used to specify a port of the instance that is to be considered an external port of the entire application. Its value is the name of this instance's port that should be externalized. The external application-level name of the port is the same as its own name on this instance. To specify a different name, use the "connection" element described below.

2.4 Property Elements within the Instance Element (optional)

The "property" element is used as a child of the "instance" element to specify configuration property values that should be configured in the worker when the application is run, prior to the application being started. Within an ***instance*** element, some examples of property (child) elements are:

```
<instance component="psd">
  <property name="size" value="17"/>
  <property name="symmetric" value="true"/>
</instance>
```

Properties can only be set if they are specified as *writable* or *initial* (meaning they can be set initially, but cannot be written during execution).

2.4.1 Name attribute (required)

The "name" attribute of a property element must match the name of a property of the specified component. I.e., it must be one of the defined configuration properties of the

component. Component specifications define properties that are common to all implementations of a component. Component implementations (workers) can also define additional properties that are specific to that implementation, but mentioning such properties will only be accepted if the selected implementation has them. Otherwise an error results.

2.4.2 Value attribute (optional – use "value" or "valueFile")

The "value" attribute is the value to be assigned to the configuration property of the worker just before being started. The attribute's value must be consistent with the data type of the property in the component specification. I.e. if the type of the property is "unsigned long", then the attribute's value must be numeric. The complete syntax of property values is described in the Property Value Syntax section below.

2.4.3 ValueFile attribute (optional – use "value" or "valueFile")

The "ValueFile" attribute is the name of a file containing the value to assign to the property. Using this attribute, rather then the "value" attribute, is convenient when the value is large, such as when the property's value is an arrays of values. When "valueFile" is used, all new lines in the file are interpreted as commas.

The complete syntax of property values is described in the Property Value Syntax section below.

2.4.4 DumpFile attribute (optional)

The "DumpFile" attribute is the name of a file into which the value of the property will be written after execution (when using the "ocpirun" utility program described below). When "DumpFile" is used, all commas in the "value" are replaced by new lines in the written file. The complete syntax of property values is described in the Property Value Syntax section below.

2.5 Property Elements within the Application Element (optional)

Property elements at the top level of an application (rather than under an ***instance*** element), represent properties of the application as a whole. They are essentially a mapping from a top-level property name to a property of some instance in the assembly.

This provides a convenient way to expose properties to the user of an application without requiring them to know the internal structure of the application.

2.5.1 Name attribute (required)

The name attribute of an application-level property is the name that users of the application will use to read, write or display the value. If the "property" attribute below is not present, then this name is also the name of the instance's property.

2.5.2 Instance attribute (required)

This attribute specifies the name of the instance that actually implements this property for the application. It is the instance that the application-level property is "mapped to".

2.5.3 Property attribute (optional)

If the application-level name of this property is not the same as the instance's property to which is it mapped, this attribute is used to specify the actual property of the instance.

2.6 Connection Elements within the Application Element (optional)

Connecting instances together in an assembly can be done one of two ways. Using the ***connect*** attribute of an instance is the simplest (described above), but cannot express all connections. This ***connection*** child element of the "application" element can be used to express all types of connections. It describes connections among ports and also with "the outside world", i.e. external to the application. The ***connection*** element has optional ***name*** and ***transport*** attributes, and ***port*** and ***external*** child elements.

An example of an application with some connections is:

```
<application done='file_write'>
  <instance component='file_read'/>
  <instance component='bias'/>
  <instance component='file_write'/>
  <connection transport="socket">
    <port instance='file_read' name='out'/>
    <port instance='bias' name='in'/>
  </connection>
  <connection>
    <port instance='bias' name='out'/>
    <port instance='file_write' name='in'/>
  </connection>
</application>
```

The first connection connects the "out" port of the "file_read" instance to the "in" port of the "bias" instance, and specifies that the connection should use the "socket" transport mechanism. The second simply connects the "out" port of the "bias" instance to the "in" port of the "file_write" instance.

2.6.1 Name attribute (optional)

This attribute specifies the name of the connection. It is only used for documentation and display purposes and has no specific other function. If it is not present, a name is assigned according to the "conn<n>" pattern, where <n> is the number of the connection in the application (0 origin). If the connection is thought of as a "wire", this is the name of the wire that is attached to various other things (instance ports and external ports).

2.6.2 Transport attribute (optional)

This attribute specifies what transport mechanism should be used for this connection. OpenCPI supports a variety of transport technologies and middlewares that convey data/messages from one instance's port to another. Normally the transport mechanism is chosen automatically based on which ones are available and optimal. This attribute allows the application to override the default transport mechanism and force the usage of a particular one. The ones supported at the time of this document update are:

pio	Programmed I/O using shared memory buffers between processes
pci	DMA or PIO over the PCI Express bus/fabric
ofed	RDMA using the OFED software stack, usually for Infiniband
socket	RDMA using TCP/IP sockets
ether	RDMA using Ethernet (link layer) frames
udp	RDMA using UDP/IP

Some of these transport mechanisms are only available if specifically installed in a system. See the system configuration guide [TBW].

2.7 Port Elements within the Connection Element (optional)

This element is used to specify a port that this connection should be attached to. The most common use of this element is to specify the consumer and producer of the connection, using a ***port*** element for each, within the same ***connection*** element. However, port elements can be used to indicate more than two ports on the same connection, when there are multiple consumers for the connection (currently not supported).

2.7.1 Name attribute (required)

This attribute specifies the name of the port that should be attached to this connection. This port name is scoped to the instance defined in the instance attribute to the connection element.

2.7.2 Instance attribute (required)

This attribute specifies the name of the instance that should be attached to this connection. This instance name is used along with the port attribute to specify the port.

3 Utility program for executing XML-based Applications: ocpirun

The simplest way to run an OpenCPI application is to describe it in an XML file (an OAS as described above), and run it using the command-line utility "***ocpirun***". This command reads the OAS file and runs the application. E.g., if the OAS was in a file named "myapp.xml", the following command would run it:

```
ocpirun myapp.xml
```

The execution ends when the application described in the OAS is "done". As mentioned above, an application is done either when all the workers signal they are "done" or when a single worker, indicated as the "done" attribute in the OAS, says it is done. The ***ocpirun*** utility also has an option to stop execution after a fixed period of time.

There are several options to ***ocpirun***, which are printed when it is executed with no arguments. When an option has a value, the value can immediately follow the option letter, or be in the next argument. The options are:

- ***-d*** *Dump all readable properties of instances when done, to standard error.*
- ***-v*** *Be verbose: print messages indicating the progress of the application*
- ***-s [<instance-name>]=<expression>***
  Set the selection expression for the named instance to the given expression. This overrides any selection attribute on the instance in the OAS. This option will apply to all instances if the "<instance-name>" is not present.
- ***-m [<instance-name>]=<model>***
  Set the required authoring model of the named instance to the given model. This creates a constraint in addition to that implied by any selection expression of the instance in the OAS. This option will apply to all instances if the "<instance-name>" is not present.
- ***-p <property-name>=<value>***
  Set the value of the indicated application-level property to the given value. This overrides any setting of the same property values in the OAS.
- ***-P [<instance-name>]=<platform-name>***
  (Upper case P) Set the platform type that the instance should run on. This option will apply to all instances if the "<instance-name>" is not present.
- ***-c [<instance-name>]=<container-name>***
  Assign the named instance to the given container. The first rcc container is named "rcc0". This option will apply to all instances if the "<instance-name>=" is not present.
- ***-t <time-to-execute-in-seconds>***
  Stop execution after this many seconds. This is useful when there is no definition of "done" for the application and thus it would otherwise run indefinitely or until ***ocpirun*** *was interrupted (e.g. control-C).*
- ***-n <number of rcc containers to create>***
  Specify the number of software/rcc containers to create to run the application. The default is 1.

- ***-l <log-level>***
  For this execution, set the OpenCPI log level to the given level. This overrides any value of the OCPI_LOG_LEVEL environment variable.

4 Property Value Syntax and Ranges

This section describes how property values are formatted to be appropriate for their data types. Property values occur in the "value" attribute of "property" elements described above, as well as when property values are specified in character strings in the APIs described below. The names of the types are those acceptable to the "type" attribute of the "property" element in the OCS (OpenCPI Component Specification).

4.1 Values of unsigned integer types: uchar, ushort, ulong, ulonglong

These numeric values can be entered in decimal, octal (leading zero), or hexadecimal (leading 0x). The limits are the typical ranges for unsigned 8, 16, 32, or 64 bits respectively, also consistent with the ranges defined in CORBA IDL.

The uchar type can also be entered as a value in single quotes, which indicates that the value is the value of the ASCII character in single quotes (as in C).

4.2 Values of signed integer types: short, long, longlong

These numeric values can be entered in decimal, octal (leading zero), or hexadecimal (leading 0x), with an optional leaving minus sign to indicate negative values. The limits are the typical ranges for signed 16, 32, or 64 bits respectively, also consistent with the ranges defined in CORBA IDL. Note that the "char" type cannot be entered directly as a signed integer, but only as a "character" as specified below.

4.3 Values of the type: char

This type is meant to represent a character, i.e. a unit of a string. In software it is represented as a "signed char" type, with the typical numeric range for a signed 8-bit value. The format of a value of this type is simply the character itself, with the typical set of "escapes" for non-printing characters, as specified in the C programming language and IDL (**`\n, \t, \v, \b, \r, \f, \a, \\, \?, \', \"`**). A series of 1-3 octal digits can follow the backslash, and a series of 1-2 hex digits can follow **`\x`**.

OpenCPI adds two additional escape sequences as a convenience for entering signed and unsigned decimal values. The sequence **`\d`** may be followed by an optional minus sign (–) and one to three decimal digits, limited to the range of -128 to 127. The sequence **`\u`** can be followed by 1 to three decimal digits, limited to the range of 0 to 255.

These escapes can also be used in a string value. Due to the requirements of the arrays and sequence values defined below, the backslash can also escape commas and braces (**`\,`** and **`\{`** and **`\}`**).

4.4 Values of the types: float and double

These values represent the IEEE floating point types with their defined ranges and precision. The values are those acceptable to the ISO C99 strtof and strtod functions respectively.

4.5 Values of the type: bool

These values represent the boolean type, which is logical true or false. The values can be (case insensitive): “true” or “1” for a true value, and “false” or “0” for a false value.

4.6 Values of the type: string

These values are simply character strings, but also can include all the escape sequences defined for the “char” type above. Due to the requirements of the arrays and sequence values defined below, the backslash can also escape commas and braces (`\,` and `\{` and `\}`).

4.7 Values in a sequence type

Values in a sequence type are simply comma-separated values. When the data type of a sequence is “char” or “string”, backslash escapes can be used when the data values include commas.

4.8 Values in an array type

When a value is a one-dimensional array, the format is the same as the sequence, with the number of values limited by the size of the array. If the number of comma-separated values is less than the size of the array, the remaining values are filled with the “null” value appropriate for the type. Null values are zero for all numeric types and the type “char”. Null values for string types are empty strings. As stated above, when a property value is placed in a file using the “valuefile” or “dumpfile” attributes of the “property” element, newlines characters in the file are replaced with commas (for “valuefile”) and vice versa for “dumpfile”. This allows array elements to be placed in files one array element per line.

4.9 Values in multidimensional types

For multidimensional arrays or sequences of arrays, the curly brace characters (`'{'` and `'}'`) are used to define a “sub-value”. For example, a sequence of 3 elements of arrays of length 3 of type ***char***, would be:

```
{a,b,c},{x,y,z},{p,q,r}
```

This would also work for a 3 x 3 array of type ***char***. Thus braces are used when an item is itself an array, recursively.

4.10 Values in struct types

Struct values are written as a comma-separated sequence of members, where each member is a member name followed by white space, followed by the member value. A struct value can be “sparse”, i.e. only have values for some members. Thus if the struct type was:

```
struct { long l[2][3]; string m2; char c; };
```

A valid value would be:

```
l {{1,3,2},{4,5,6}}, c x
```

This value would not have a value for the “m2” member.

5 API for executing XML-based Applications in C++ programs

Although XML applications can easily be executed using the ***ocpirun*** command, there are cases where more programmatic and/or dynamic creation or control of the XML-based application is required. Here are examples of when ***ocpirun*** may be inadequate, and require using the C++ "Application Control Interface" API (called ACI below).

- The contents of the application XML (OAS) need to be constructed programmatically.
- The C++ (main) program needs to directly connect to the ports of the running application (see ***external ports*** below).
- The XML-based application needs to be run repeatedly (perhaps with configuration changes) in the same process.
- Some of the attributes of the XML application need to be dynamically overridden by the C++ application.

We use the term "control application" to describe the C++ application using this interface. In all examples below, the namespace prefix "OA" is used as an abbreviation of the actual namespace of the API: "OCPI::API::", i.e. assuming:

```
namespace OA = OCPI::API;
```

The API for executing XML-based applications is based primarily on one C++ class: **OCPI::API::Application**. It is constructed by referring to the OAS and has various lifecycle control member functions. It is well suited to being constructed with automatic storage (on the stack) and using the implicit destruction at the end of the block. A simple example using this API, assuming the OAS is in the file "myapp.xml", is:

```
{
  OA::Application app("myapp.xml");
  app.initialize(); // all resources allocated
  app.start();      // start execution
  app.wait();       // wait until app is "done"
}
```

5.1 Class OCPI::API::Application

This class represents a running application, with a simple lifecycle. It has constructors and destructors suitable for automatic storage, methods for getting and setting configuration properties, and methods for directly communicating with the external ports defined in the application.

5.1.1 **OA:: Application::Application** constructors

There are two constructors for this class that differ only in the type of the first argument. If it is "const char *", it is a filename containing the OAS. If it is a "const std::string&", it is the OAS itself as a string. The second argument is a parameter array, of type "const OA::PValue *". It defaults to NULL (no parameters).

The constructor searches the available component libraries as specified in the OCPI_LIBRARY_PATH environment variable, and chooses an implementation from those available in the libraries for each instance in the OAS. Resources are ***not***

allocated (no loading or instantiating or configuring or connecting is performed). When the constructor returns successfully (without exception), the OAS is valid and implementations have been found and selected for all instances in the OAS. Here are the two constructors:

```
class Application {
  Application(const char *file, OA::PValue *params = NULL);
  Application(std::string &oas, OA::PValue *params = NULL);
};
```

The “params” argument is used to provide additional constraints on the selection of implementations and the assignment to containers, in addition to providing more property values. All these values could be specified in the OAS, but this allows the OAS to remain constant while various aspects of the execution are overridden or augmented.

The “property”, “selection”, “model”, and “container” parameters perform the same function as the “-p”, “-s”, “-m”, and “-c” flags to the ***ocpirun*** utility program. Their values are strings that specify a parameter relative to a particular instance. An example is:

```
{
  OA::PValue params[] = {
      PVString(“model”, “psd1=rcc”),
      PVString(“selection”, “filter=snr<40”),
      PVString(“property”, “filter=mode=6”),
      PVEnd
  };
  OA::Application app(“myfile.xml”, params);
  app.initialize(); // all resources allocated
  app.start();      // start execution
  app.wait();       // wait until app is “done”
}
```

The syntax of the OA::PValue class is described below. Except for the “property” parameter, if there is no instance (followed by equal sign), the parameter applies to all instances. E.g.:

```
OA::PValue params[] = {
  PVString(“model”, “rcc”),
  PVString(“selection”, “filter=snr<40”),
  PVString(“property”, “filter=mode=6”),
  PVEnd
  };
}
```

would specify that all instances should use the “rcc” model.

5.1.2 **OA:: Application::initialize** method

This method initializes the application by allocating all necessary resources and loading, creating, configuring and connecting all workers necessary to run the application. When this method returns the application is “ready to run”. Any errors that might occur when allocating resources, loading code, instantiating workers, configuring workers or connecting workers, will have happened via exceptions before this method returns.

```
class Application {
  initialize();
};
```

5.1.3 **OA:: Application::start** method

This method starts the application by starting all the workers in the OA::Application. When this method returns the application is running.

```
class Application {
  start();
};
```

5.1.4 **OA::Application::stop** method

This method suspends execution of the application. When the method returns the application is no longer executing. Properties may be queried (and should not be changing) after the application is suspended. Some workers do not implement this operation (they are not suspendable), and if so an exception is thrown. When the application can be successfully stopped, it can be resumed by again using the **start** method described above.

```
class Application {
  void stop();
};
```

5.1.5 **OA:: Application::wait** method

This method blocks the caller until the application is done: when all the workers are done or when the worker indicated by the "done" attribute in the OAS is done. The single argument indicates how long to wait in microseconds. If the value is zero, the wait will not timeout. The return value is true when the timeout expired, and false when the application was "done".

```
class Application {
  bool wait(unsigned timeout_us);
};
```

5.1.6 **OA:: Application::finish** method

This method performs various functional (not cleanup) actions when the application is "done". It can be called after "wait" returns, whether timeout or not. Among other things, this is required to perform the "dump" action indicated by the "***-d***" option to ***ocpirun***.

```
class Application {
  void finish();
};
```

5.1.7 **OA::Application::getProperty** method

This method get's an instance's property value by name, returning the value in string form into the std::string whose reference is provided. It should be used in preference to the OA::Property class below, when performance is not important, since although it has higher overhead internally, it is simpler than using OA::Property.

There are two overloaded versions of this method.

If the property being accessed is *not* a top-level property defined for the application as a whole, the "name" argument is of the form "**<instance-name>.<property-name>**".

If there is no property with the given name, or the property is not readable, or some other error occurs reading the property value, an exception is thrown.

```
class Application {
  void getProperty(const char *name, std::string &value);
  bool getProperty(unsigned ordinal, std::string &value);
};
```

The second method is used to retrieve the property's name and value by ordinal. It is useful to retrieve all property values without knowing their names. The return value is true if the ordinal is valid. Thus a simply loop can retrieve all properties:

```
std::string name, value;
for (unsigned n = 0; app.getProperty(n, name, value); n++)
  std::cout << name << ":" << value << std::endl;
```

5.1.8 **OA::Application::setProperty** method

This method sets a by name, taking the value in string form, which is then parsed and error checked according to the data type of the property. It should be used in preference to the OA::Property class below, when performance is not important, since although it has higher overhead internally, it is simpler than using OA::Property.

The "name" argument is of the form "**<instance-name>.<property-name>**".

If the value cannot be parsed for the appropriate type, or there is no property with the given name, or the worker itself does not accept the property setting, an exception is thrown.

```
class Application {
  void setProperty(const char *name, const char *value);
};
```

5.1.9 **OA::Application::getPort** method

This method is used when the C++ program wants to directly connect to an "external" port of the application. Such a connection is external to the application as defined in the OAS (via the "external" attribute of an "instance" element, or an "external" child element of a "connection" element). This allows the C++ program to directly send and receive messages to/from the application (actually to/from some port of some instance in the application).

An optional PValue list is provided to each side of the connection in order to provide configuration information about the connection. The producer or consumer type of the created ExternalPort object is implicitly opposite from the role of the external port. E.g. if the external port is an output port, then the ExternalPort object acts as an input port on which to receive messages. This method returns a reference to an ExternalPort object that is used by the control application to, itself, produce or consume data.

```
class ExternalPort;
class Application {
  ExternalPort &getPort(const char *externalName,
                        const PValue *myProperties = NULL,
                        const PValue *extProperties = NULL);
};
```

If the connection cannot be made or the PValue lists are invalid, an exception is thrown.

The possible PValue types for these external connections are the same as the connect method of the Port class below [put them here and refer back to them..].

5.2 Class OA::ExternalPort

This is the class of objects that represent a communication endpoint in the control application itself, used to communicate with external ports. They are owned by the OA::Application object. They are not deleted directly, but are only destroyed when the OA::Application is destroyed.

5.2.1 **OA::ExternalPort::getBuffer** method

This method is used to retrieve the next available buffer on an external port. It returns a pointer to an ExternalBuffer object, or NULL if there is no buffer available. Thus it is a non-blocking I/O call used by the control application. For external ports acting in the producer role, the returned buffer is a buffer to fill with a message to send. For external ports acting in the consumer role, the returned buffer is a buffer that contains the next message that can be received/processed by the application. When the control application is done with the buffer, it calls the "**put**" method (for sending/producing) or the "**release**" method (for discarding input buffers). In addition to returning the buffer object, the getBuffer method also returns (as output arguments by reference), the data pointer into the buffer and the length of the message (for input) or buffer (for output).

There are actually two overloaded getBuffer methods, for the two directions. The first, for getting a buffer filled with an incoming message, also returns the metadata for message (opCode and endOfData) in separate by-reference output arguments.

```
class ExternalBuffer;
class ExternalPort {
  // Input: get buffer filled with next incoming message
  ExternalBuffer *getBuffer(uint8_t &data,
                            uint32_t &length,
                            uint8_t &opCode,
                            bool &endOfData);
  // Output: get buffer to fill with next outgoing message
  ExternalBuffer *getBuffer(uint8_t &data,
                            uint32_t &length);
};
```

5.2.2 **OA::ExternalPort::endOfData** method

This method, used only when the role of the external port is producer, is used to indicate that no more messages will be send on this connection. This propagates an out-of-band indication across the connection to the port. Note that this indication can

also be made in the ExternalBuffer::put() method below if the message being sent is the last message to be sent. This latter method may be more efficient, since the out-of-band indication can be carried with the message, rather than by itself.

```
class ExternalPort {
  void endOfData();
};
```

5.2.3 **OA::ExternalPort::tryFlush** method

This method, used only when the role of the external port is producer, is used to attempt to "move data out the door", when messages are locally buffered in this single-threaded non-blocking environment. The return value indicates whether there are still messages locally buffered that will require further calls to tryFlush.

```
class ExternalPort {
  bool tryFlush();
};
```

Note this is only required in single-threaded environments.

5.3 Class OA::ExternalBuffer

This is the class of objects that represent buffers attached to (owned by) external ports. They are returned (by pointer return value) from the ExternalPort::getBuffer methods, and given back to the external port via the "**put**" method (for output) or the "**release**" method for input.

5.3.1 **OA::ExternalBuffer::release** method

This is the method used to discard an input buffer after it has been processed/consumed by the control application.

```
class ExternalBuffer {
  void release();
};
```

5.3.2 **OCPI::ExternalBuffer::put** method

This is the method used to send an output buffer after it has been filled by the control application. The arguments specify the metadata associated with the message: (1) the length in bytes of valid message data, (2) (optional) the opcode of the message, and (3) (optional) whether it is the last message to be sent (if that fact is known at the time of the call).

```
class ExternalBuffer {
  void put(uint32_t length,
           uint8_t opCode = 0, bool endOfData = false);
};
```

5.4 Class OA::Property

This is the class of objects that represent a runtime accessor for a property. They are normally created with automatic storage (on the stack) and simply cache the necessary information to very efficiently read or write property values. The control application that

uses this class is responsible for creating and deleting the objects, although typical usage is automatic instances that are automatically deleted.

5.4.1 **OA::Property::Property** constructor method

This constructor initializes the Property object such that it is specific to the application and specific to a single named property of that application.

```
class Property {
  Property(Application &app, const char *name);
};
```

The name argument specifies the property the same as the getProperty method described above. Typical usage would be:

```
{
  OA::Application app("myapp.xml");
  app.initialize();
  OA::Property
      freq(w, "frequency"),
      peak(w, "peak");
  app.start();
  freq.setFloatValue(5.4);          // set this during execution
  float p = peak.getFloatValue();   // get this during execution
  app.wait();
}
```

The "**set**" and "**get**" methods are all strictly typed. They cannot be overloaded since overloading of integral types in C++ does not prevent truncation.

This same class is used in the more detailed ACI classes described below. In particular, there is another constructor for this class based on a Worker object:

```
class Property {
  Property(Worker &worker, const char *name);
};
```

Beyond the fact that it is based on a worker rather than an application, the constructed Property object is used with all the same methods.

5.4.2 **OA::Property::set{Type}Value** methods

There is a **set** method for each data type that a property is allowed to have. The **set** methods are strongly typed and individually named to avoid the unintended consequences of numerical type conversions of the C++ language. If the wrong **set** method is used for a property (i.e. setULong for a property whose type of Float), an exception is thrown. If the string in **setStringValue** is longer than the worker property's maximum string length, an exception is thrown.

```
class Property {
  void setBoolValue(bool val);
  void setCharValue (int8_t val);
  void setDoubleValue (double val);
  void setFloatValue (float val);
  void setShortValue (int16_t val);
  void setLongValue (int32_t val);
  void setUCharValue (uint8_t val);
  void setULongValue (uint32_t val);
  void setUShortValue (uint16_t val);
  void setLongLongValue (int64_t val);
  void setULongLongValue (uint64_t val);
  void setStringValue (const char *string);
};
```

5.4.3 **OA::Property::get{Type}Value** methods

There is a **get** method for each data type that a property is allowed to have. The **get** methods are strongly typed and individually named to avoid the unintended consequences of numerical type conversions of the C++ language. If the wrong **get** method is used for a property (i.e. getULong for a property whose type of Float), an exception is thrown. If the string buffer in **getStringValue** is not long enough to hold the worker property's current string value, an exception is thrown. If there is an error accessing the worker's property value, an exception is thrown.

```
class Property {
  bool getBoolValue();
  int8_t getCharValue();
  double getDoubleValue();
  float getFloatValue();
  int16_t getShortValue();
  int32_t getLongValue();
  uint8_t getUCharValue();
  uint32_t getULongValue();
  uint16_t getUShortValue();
  int64_t getLongLongValue();
  uint64_t getULongLongValue();
  void getStringValue(char *string, unsigned length);
};
```

5.4.4 **OA::Property::set{Type}SequenceValue** methods

There is a set sequence method for each property data. The set sequence methods are strongly typed and individually named. If the wrong set sequence method is used for a property (i.e. setULongSequence for a property whose type of Float), an exception is thrown. If any of the strings in setStringValueSequence is longer than the property's maximum string length, an exception is thrown. It the number of items in the provided sequence is greater than the maximum sequence or array length of the property, an exception is thrown. If there is an error accessing the property value, an exception is thrown.

```
class Property {
  void
      setBoolSequenceValue(bool *vals, unsigned n),
      setCharSequenceValue(int8_t *vals, unsigned n),
      setDoubleSequenceValue(double *vals, unsigned n),
      setFloatSequenceValue(float *vals, unsigned n),
      setShortSequenceValue(int16_t *vals, unsigned n),
      setLongSequenceValue(int32_t *vals, unsigned n),
      setUCharSequenceValue(uint8_t *vals, unsigned n),
      setULongSequenceValue(uint32_t *vals, unsigned n),
      setUShortSequenceValue(uint16_t *vals, unsigned n),
      setLongLongSequenceValue(int64_t *vals, unsigned n),
      setULongLongSequenceValue(uint64_t *vals, unsigned n),
      setStringSequenceValue(const char **string, unsigned n);
};
```

5.4.5 **OA::Property::get{Type}SequenceValue** methods

There is a get sequence method for each data type that a worker's property is allowed to have. The get sequence methods are strongly typed and individually named. If the wrong get sequence method is used for a property (i.e. getULongSequenceValue for a property whose type of Float), an exception is thrown. If the string buffers in getStringSequenceValue (specified by the maxStringLength argument) are not long enough to hold the worker property's current string values, an exception is thrown. If there is an error accessing the worker's property value, an exception is thrown.

```
class Property {
  void
      getBoolSequenceValue(bool *vals, unsigned n),
      getCharSequenceValue(int8_t *vals, unsigned n),
      getDoubleSequenceValue(double *vals, unsigned n),
      getFloatSequenceValue(float *vals, unsigned n),
      getShortSequenceValue(int16_t *vals, unsigned n),
      getLongSequenceValue(int32_t *vals, unsigned n),
      getUCharSequenceValue(uint8_t *vals, unsigned n),
      getULongSequenceValue(uint32_t *vals, unsigned n),
      getUShortSequenceValue(uint16_t *vals, unsigned n),
      getLongLongSequenceValue(int64_t *vals, unsigned n),
      getULongLongSequenceValue(uint64_t *vals, unsigned n),
      getStringSequenceValue(const char **string,
                             unsigned n,
                             char *buf,
                             unsigned maxStringSpace);
};
```

5.5 Class OA::PValue: named and typed parameters

This is the class of objects that represent a strongly typed name/value pair, and is always used as a member of a null-terminated array of such objects. Its usage is typically to provide a pointer to an array of PValue structures, usually statically initialized. There are derived classes (of PValue) for each supported data type, which is

the same set of types supported for worker configuration properties and protocol operation arguments in the metadata associated with workers. Thus for each supported scalar data type, the name of the derived class is P{*type*}, where *type* can be any of:

Bool, Char, Double, Float, Short, Long, UChar, ULong, UShort, LongLong, ULongLong, or String.

The corresponding C++ data types are:

bool, char, double, float, int16_t, int32_t, uint8_t, uint32_t, uint16_t, int64_t, uint64_t, char *.

Common usage for static initialization is to declare a PValue array and initialize it with typed values and terminate the array with the symbol PVEnd, which is a value with no name, e.g.:

```
PValue pvlist[] = {
  PVULong("bufferCount", 7),
  PVString("xferRole", “active”),
  PVULong("bufferSize", 1024),
  PVEnd};
```

Note that PValue objects are used named and typed parameters to runtime APIs, and are in fact unrelated to “properties” except they share data types.

6 Classes for directly creating and executing applications

When using the XML applications (OAS) does not provide sufficiently detailed control of the execution of an OpenCPI application, the follow aspects of the ACI can be used.

This C++ interface creates and/or uses the following objects (and classes) to run and control applications:

Containers: execution environments/processors where workers might execute

ContainerApplications: collection of workers on one container that are together performing some of the work of the application

Artifacts: binary files like DLL/.so files or bitstream files for FPGAs from which workers are instantiated

Workers: instantiated implementations of components from binary code in Artifacts

Ports: attachment points on workers that produce or consume data, communicating with other workers in the application, or as input/output to and from the application as a whole

ExternalPorts: data sources and sinks in the control application itself

Properties: configuration properties of workers that may be read or written at runtime

The ***ContainerApplication*** class is used as the lifecycle object that owns all the ***workers*** instantiated on a container for the execution of the (component-based) application. Thus the interface described below only has lifecycle (create and destroy) control over the ***containerapplication*** classes and the ***container*** classes. All these classes (Container, ContainerApplication, Artifact, Worker, Port, Property) are in the OCPI::API C++ namespace.

Container objects represent an execution environment for ***workers***. ***Container*** objects for executing software workers (RCC etc.) can be within the process of the control application using this API, or can represent some other execution environment (other process or other processing device) that is available to the OpenCPI implementation and installation (system). At the level of abstraction of this *application control interface*, the caller knows the name of the component to be executed, and then the *specific implementation in a specific artifact* is found by OpenCPI by looking for ***artifacts*** in component ***libraries*** that incorporate implementations of the named ***component***.

Thus the typical sequence for using this Application Control Interface is:

- Find ***containers*** on which the ***components*** of the application should execute.
- Establish an ***application*** context on each ***container*** in which workers can be created incrementally, and removed as a group based on that context.
- Create the ***worker*** runtime instances based on ***artifacts*** that are found by OpenCPI in some component library in the OpenCPI component library path, based on the type of container.
- Connect the ***ports*** of the ***workers:*** within the ***containers***, between the ***containers***, and to the control application itself (via "***ExternalPorts***").
- Start the ***workers*** using the OpenCPI "start" operation on the ***application*** objects.

At this point the overall *component application* is running and the *control application* can interact with the running *component application* by a combination of (1) sending and receiving messages on external ports attached to worker ports, (2) reading runtime property values of workers to obtain scalar results and status, and (3) setting runtime property values to control/parameterize/modify the execution of the workers, and (4) start and stop individual workers for various purposes including debugging.

When the *control application* is finished using the *component application*, it can simply destroy all the application contexts on all containers (or simply exit, since they will all be automatically destroyed anyway).

The purpose of this API is to programmatically assemble an application by:

- instantiating and initializing **Workers** (specific component implementations) from **Artifacts** loaded into **Containers**
- connecting the Workers' **Ports** to each other or to the application control code (the caller of this interface)
- set initial configuration **Properties** of workers
- start the execution of the **Workers** in the **Application**
- during execution, possibly pause, resume, read and write dynamic **property** values of the **workers** in the application
- tear down the **application** (all the **workers** on all the **containers** used).

Each of these object classes and their methods (called "member functions" in C++) will be defined below. This interface is for cases where the controlling application (the conventional application that runs and controls the component-based OpenCPI application) wants direct and fine-grained control over the application and the workers and ports in the application. The XML-based utility program (***ocpirun***) and related C++ interfaces described earlier, offer simpler mechanisms for running OpenCPI applications that are more "fully scripted" so that no such C++ code is required.

6.1 Requirements for all classes in this API

- Clear lifecycle
- Use references for things not under the callers lifecycle control
- Use the OCPI::API namespace (generally abbreviated OA::) to avoid any namespace collisions with other user or library code

6.2 Classes in the Application Control Interface

The classes in the ACI are listed in the order you would use them in a typical program. All are in the OCPI::API C++ namespace. It is common practice to use an abbreviation for this namespace via the C++ statement:

```
namespace OA = OCPI::API;
```

All the descriptions below will use the OA abbreviation.

6.2.1 Class OA::ContainerManager

This class represents a singleton object in the process address space that is mostly used via static methods. It is a "bootstrapping" class in that it allows the control application to "get started" in using the ACI's objects. It provides access to containers that will be used as places to run workers. Since ***Container*** objects represent places to run workers, and they are commonly one-per-processing-node in the system, the ***ContainerManager*** can be thought of as the owner and manager of all ***containers*** accessible from the control application.

The ***ContainerManager*** has a number of ways to find out about containers in the system. Some of them may be internally configured, while others may be discoverable by looking at available hardware and available drivers for processing hardware in the system. The ***ContainerManager*** may be also explicitly informed of the existence of a container.

6.2.1.1 OA::ContainerManager::find

This *static* method is used to find a container, usually of a particular type, but sometimes a specific container for a specific processor. From the point of view of the caller, all containers behave the same, but control applications may need to find certain types of container to run certain types of workers. The arguments of **find** allow the caller to narrow down the type of container it is looking for, in order to run certain workers. There are two common scenarios:

- Find a container to run a particular worker.
- Find a container and then find a worker that will run on that container.

There are a number of ways of identifying a container, and theoretically a container can magically execute a wide range of implementation types. However, the two primary aspects are authoring model (RCC, HDL, etc.), and a specific piece of hardware (platform). Secondarily, there might be specific support for tool chains and operating environments. The ***find*** method has two overloaded versions. One simply has three arguments: the authoring model, the specific name of the container, and a PValue list for more specific options. The last two can be NULL for "don't care". The second method has an argument that is a property list using the PValue class that simply enumerates the characteristics of the container being sought.

```
class ContainerManager {
  static Container *
      find(const char *model,
           const char *which = NULL,
           const PValue *list = NULL);
      find(const PValue *list);
};
```

The return value will be NULL if no container can be found. Containers come into existence by being automatically discovered during the first call to the find method. There are many more complex ways that containers might be created but they are out of scope here. The default is that all container drivers that are linked with the OpenCPI library are used to discover and create available containers. Software containers are created on demand by name. Thus they are always "found" since they will be created if no one with the given name exists.

*6.2.1.2 **OA::ContainerManager::shutdown***

This *static* method simply shuts down all containers and thus removes any resources associated with all containers. It is only used to ensure that no resources associated with containers are retained. It returns the OpenCPI subsystem to its initial state (after static construction). This method is only needed when resources should be recovered *before* the program exits. Normally a program can simply exit and OpenCPI's container resources will be automatically recovered as appropriate.

```
class ContainerManager {
  static void shutdown();
};
```

6.2.2 Class OA::Container

This is the class of objects that represent execution environments for workers. It manages application contexts.

*6.2.2.1 **OA::Container::createApplication** method*

This method creates an application context object in which workers will be created. It can be deleted when all of the workers created with it are no longer needed. Note that the class of the created object is OCPI::API::ContainerApplication. Thus this object is *scoped to the container* that created it and represents the parts of the overall application that reside in this container. The *OCPI::API::Application* class (*not* in the OA::Container namespace) is the application object that spans multiple containers (which is supported for XML applications).

```
class Container {
  ContainerApplication *
      createApplication(const char *name = NULL,
                        const PValue *params = NULL);
};
```

The return value of this method is a newly created object that may be deleted by the caller when it is no longer needed (or by using, e.g. std::auto_ptr<>). This is only necessary if the container's resources must be recovered before program exit. The

name argument is useful in certain debugging/tracing scenarios, and the *params* argument specifies parameters of the application. Both are optional.

The only OA::PValue parameter that is defined for this method is a string-valued parameter called “package”. This supplies the default package prefix when the createWorker method (see below) is called with a component-name without a package prefix. E.g.:

```
PValue pvlist[] = {PVString("package", “com.mycorp.mylib”), PVEnd};
ContainerApplication *app = cont->createApplication(“myapp”, pvlist);
```

If this parameter is not set, the default package prefix is “local”.

6.2.3 Class OA::ContainerApplication

This is the class of objects that represent application contexts on a specific container. It is the owner of the workers it creates. If multiple applications are using a container, different objects of this class will exist in the container.

*6.2.3.1 **OA::ContainerApplication::createWorker** methods (2 overloaded)*

These two methods are used to create a worker (instance) in an application context in a container. The simpler form is given the name of this worker instance within this application (nameWithinApp), and the name of the component that the worker should implement, followed a list of initial configuration property values. This version of createWorker searches the artifact libraries indicated by the OCPI_LIBRARY_PATH environment variable (containing colon-separated directory names), looking for an artifact containing a worker implementation that will execute on the container.

The “component” argument can either be a fully qualified name that includes the package prefix (e.g. “com.mycorp.mylib.func”), or a simple name (“func”). In the latter case the package prefix set in the “createApplication” method described above is used as the package prefix.

The second more complicated version has two more arguments at the beginning of the argument list to specify an artifact file, and a parameter list for using the artifact. It also has an extra argument to identify a particular instance in the artifact – when the artifact has fixed, static instances (like FPGA bitstreams or some statically linked DSPs).

Typically, in a software artifact like a shared object/dynamic library, the worker implementation will support dynamic creation of as many worker instances as are required in the applications running in the container. However, some artifacts are statically configured with a specific set of named instances and thus the instance must be identified.

```
class Worker;
class Connection;
class ContainerApplication {
  Worker &createWorker(const char *nameWithinApp,
                       const char *component,
                       const PValue *properties = NULL,
                       const PValue *params = NULL,
                       const Connection *conns = NULL);
  Worker &createWorker(const char *nameWithinApp,
                       const char *component,
                       const PValue *properties = NULL,
                       const PValue *params = NULL,
                       const Connection *conns = NULL);
};
```

Note that the return value is a reference to a worker. The caller does not own this object and cannot delete it. The *nameWithinApp* argument simply identifies this worker within the application (i.e. when there are two workers in the application that perform the same function, they are both based on the same component, but will have different names within the application). The *component* argument identifies the component function that this worker will perform. This is used to find the implementation in some artifact in some library indicated in the OCPI_LIBRARY_PATH environment variable.

The *properties* argument contains instantiation property settings of the worker instance (properties that are *writable* or *initial*). The *params* argument provides instantiation parameters for the container to use (non are currently defined). The *conns* argument indicates an array of port connections that will be made later for this worker, to help the library search process choose the best component. These connections are simply defined by three strings: the port name on this worker, the component name of the "other" worker, and the port name of the "other" worker. The array of connection structures is terminated with a null port name.

*6.2.3.2 **OA::ContainerApplication::start** method*

This method starts all the workers in the OA::ContainerApplication. This is a convenience so that each worker doesn't have to be individually started.

```
class ContainerApplication {
  start();
};
```

*6.2.3.3 **OA::ContainerApplication::~ContainerApplication** method*

When the workers created in this application context are no longer needed, this context object can be deleted, which will in turn destroy all workers within it. This is only necessary if resources need to be recovered prior to program exit.

```
class ContainerApplication {
  ~ContainerApplication();
};
```

6.2.4 Class OA::Worker

This is the class of objects that represent worker instances (objects instantiated based on component implementations in artifacts). They are owned by the OA::ContainerApplication objects. They are not deleted directly, but are only destroyed when the OA::ContainerApplication is destroyed.

*6.2.4.1 **OA::Worker::getPort** method*

This method retrieves a reference to a port object representing one of the ports of the worker. It takes the string name of the port and returns a reference to that port. An exception is thrown if there is no port with that name.

```
class Port;
class Worker {
  Port &getPort(const char *name
                const PValue *params = NULL);
};
```

The *params* argument specified port parameters.

*6.2.4.2 **OA::Worker::setProperty** method*

This method sets a worker's property by name, providing the value in string form, which is then parsed and error checked according to the data type of the property. It should be used in preference to the OCPI::Property class below, when performance is not important. It is simpler, but has much higher overhead internally. If the value cannot be parsed for the appropriate type, or there is no property with the given name, or the property is not writable, or the worker itself does not accept the property setting, an exception is thrown.

```
class Worker {
  void setProperty(const char *name, const char *value);
};
```

*6.2.4.3 **OA::Worker::start** method*

This method starts the execution of the worker. It will continue to run until the "stop" method is called, the "release" method is called or until the application is destroyed.

```
class Worker {
  void start();
};
```

*6.2.4.4 **OA::Worker::stop** method*

This method suspends execution of the worker. When the method returns the worker is no longer executing. Properties may be queried (and volatile properties should not be changing) after the worker is suspended. Some workers do not implement this operation (they are not suspendable), and if so an exception is thrown. Workers that are suspended can be resumed by again using the **start** method described above.

```
class Worker {
  void stop();
};
```

6.2.4.5 *OA::Worker::getProperty method*

This method retrieves the name and stringified value of a property identified by its ordinal. This makes it very easy to retrieve the properties of a worker without knowing their names or types. Property ordinals start at zero. The value retrieved into the referenced std::string is converted from its native type into a string. If the ordinal is invalid (greater or equal to the number of available properties), the return value is false..

```
class Worker {
  bool getProperty(unsigned ordinal,
                   std::string &name,
                   std::string &value);
};
```

Here is an example code that prints all a worker's properties using C printf:

```
OA::Worker &w;
std::string name, value;
for (unsigned n = 0; w.getProperty(n, name, value); n++)
  printf("Property %u has name \"%s\" and value \"%s\"\n",
         name.c_str(), value.c_str());
```

6.2.4.6 *OA::Worker::setProperties method*

This method sets multiple configuration property values at once, although this method is not as fast as the methods using the **Property** class below. It takes a PValue list that expresses the names, types, and values of the properties to be set.

```
class Worker {
  void setProperties(const PValue *props);
};
```

6.2.4.7 *OA::Worker::beforeQuery method*

This method warns the worker that a group of properties is about to be read, and requests that any work required to give them consistent values should be performed. Properties that are affected by this method return "true" to the "readSync" method on the associated Property object. This method is only used when it is important that the values of a group of properties be consistent with each other.

```
class Worker {
  void beforeQuery();
};
```

6.2.4.8 *OA::Worker::afterConfigure method*

This method tells the worker that a group of property changes have been made (via setProperty, or the setting methods on the Property objects), and that the worker should now consider all the values are a coherent update. Properties that are affected by this method return "true" to the "writeSync" method on the associated Property object. This method is only used when the worker is specified as requiring it.

```
class Worker {
  void afterConfigure();
};
```

6.2.5 Class OA::Port

This is the class of objects that represent a worker's ports. They are owned by the OA::Worker objects. They are not deleted directly, but are only destroyed when the OA::ContainerApplication is destroyed.

*6.2.5.1 **OA::Port::connect** method*

This method makes a connection between two ports of different workers that may or may not be in different containers. It configures the connection and enables messages to flow from one to the other. If the two ports have the same role (both are producer/user/client or both are consumer/provider/server), an exception is thrown. The port on which this method is invoked can be either a producer/user/client or a consumer/provider/server. A PValue list is provided to each side of the connection in order to provide configuration information about the connection that is not the default. The possible PValue types for connections (for either side) are:

PValue Name	Data Type	Default	Description
bufferCount	ULong	2	The number of buffers to allocate for this end of the connection
bufferSize	ULong	From Worker	The number of bytes per buffer
xferRole	String	optimized for "push"	Specify DMA transfer roles when the connection is crossing a fabric or bus. Can be: "passive", "active", "flowcontrol", or "activeonly"
transport	String	depends	Specify the message transport mechanism, e.g.: "pio", "pci", "socket", "ether", "udp", "ofed"

```
class Port {
  void connect(Port &otherPort,
               const PValue *myParams = NULL,
               const PValue *otherParams = NULL);
};
```

*6.2.5.2 **OA::Port::connectExternal** method*

This method makes a connection between ports of a worker in some container, and the control application itself. It allows messages to flow between the worker port and the control application. A PValue list is provided to each side of the connection in order to provide configuration information about the connection. The producer or consumer type of the created ExternalPort object is opposite from the role of the worker port object on which the method is called. The possible PValue types for these external connections are the same as the connect method above. This method returns a reference to an ExternalPort object that is used by the control application to, itself, produce or consume data.

```
class ExternalPort;
class Port {
  ExternalPort&
      connectExternal(const char *externalName,
                      const PValue *myParams = NULL,
                      const PValue *extParams = NULL);
};
```

If the connection cannot be made or the PValue lists are invalid, an exception is thrown.

6.3 An Example of Using the ACI

This example uses an RCC worker called “copy” that copies its input to its output, and is compiled, linked and in a directory accessible via OCPI_LIBRARY_PATH. The control application creates the worker, creates an external port to talk to its input port, and another external port to talk to its output port, sends a “hello” message, and expects to receive a “hello” message back.

```
#include “OcpiApi.h”
namespace OA = OCPI::API;
int main(int, char**) {
  OA::Container &c = OA::ContainerManager::find(“rcc”);
  OA::ContainerApplication *app = c.createApplication();
  Worker &w = app->createworker(“mycopy”, “copy”);
  Port
    &win = w.getPort(“in”),
    &wout = w.getPort(“out”);
  ExternalPort
    &myIn = win.connectExternal(“aci_out”),
    &myOut = wout.connectExternal(“aci_in”);
  w.start();
  uint8_t opcode, *idata, *odata;
  uint32_t ilength, olength;
  bool end;
  ExternalBuffer *myOutput = win.getBuffer(odata, olength);
  assert(myOutput && olength >= strlen(“hello”) + 1);
  strcpy(odata, “hello”);
  myoutput->put(strlen(“hello”) + 1);
  ExternalBuffer *myInput =
    wout.getBuffer(idata, opcode, ilength, end);
  assert(myInput && opcode == 0 &&
         ilength == strlen(“hello”) + 1 && !end &&
         strcmp(idata, “hello”);
  return 0;
}
```

7 Glossary

Component Application – A component application is a composition or assembly of components that as a whole perform some useful function.

Control Application – A control application is the conventional application that constructs and runs component applications.

Configuration Properties – Named value locations of a worker that may be read or written. Their values indicate or control aspects of the worker's operation. Reading and writing these property values may or may not have side effects on the operation of the worker. Each worker (component implementation) may have its own, possibly unique, set of configuration properties.

Control Operations – A fixed set of control operations that every worker has. The control aspect is a common control model that allows all workers to be managed without having to customize the management infrastructure software for each piece of IP, while the aforementioned configuration properties are used to specialize components. The most commonly used are "start" and "stop".

Worker – A concrete implementation (and possibly runtime instance) of a component, written according to an authoring model.

Authoring Model – A concrete implementation (and possibly runtime instance) of a component, written according to an authoring model.