OpenCPI Generic Authoring Model Reference

(AMR)

Revision History

Revision	Description of Change	Date
1.01	Creationt from a combination of the SCA CP289 specification and the original OpenCPI WIP (Worker Interface Profile) document	2010-5-01
1.1	Add clarity to parameter and read-only properties, a general sync with current SW, and a conversion to the LibreOffice odt format from the previous Microsoft Word format.	2014-4-01

Table of Contents

1 References

This document depends on several others.

Table 1 - Table of Reference Documents

Title	Published By	Link
XML	W3C	Public URL: http://www.w3.org/TR/xml
OpenCPI Technical Summary	OpenCPI.org	Public URL:

2 Introduction

The purpose of this document is to specify the concept of an OpenCPI authoring model, and to define aspects common to all authoring models. This document is a prerequisite to the other documents that specify and use the individual authoring models. It specifies the concepts, lifecycle states and related operations, and XML metadata used and manipulated by OpenCPI tools and OpenCPI component developers.

We use the term *generic* in this context to mean common to all authoring models. We use the term “component” to encompass the functionality and abstract interface aspects. We use the term “worker” to mean a particular implementation of a component written (or “authored”) using some programming language source code.

We define an authoring model casually as “a way to write a worker”. A primary goal is to support different processing technologies such as General Purpose Processors (GPPs), Field-Programmable Gate Arrays (FPGAs), Digital Signal Processors (DSPs), or Graphics Processors (GPUs).

Since there is no one language or API that allows all these processing technologies to be utilized with efficiency and utilization comparable to their “native” languages and tool environments, we define a set of “authoring models” that achieve native efficiency with sufficient commonality with other models to be able to:

- Implement an OpenCPI worker for a class of processors in a language that is efficient and natural to users of such a processor
- Be able to switch (replace) the authoring model and processing technology used for a particular component in a component-based OpenCPI application without affecting the other components of the application.
- Combine workers (component implementations) into an application using a multiplicity of authoring models and processing technologies.

An OpenCPI Authoring model consists of these specifications:

- An XML document structure/schema/definition to describe the aspects of the implementation that are specific to the authoring model being used and needed by tools and runtime infrastructure software.
- Three sets of programming language interfaces used for interactions between the worker itself and its environment:
  1. Control and configuration interfaces for run-time lifecycle control and configuration, sometimes referred to as “the control plane”.
  2. Data passing interfaces used for workers to consumer/produce data from/to other workers in the application (of whatever model on whatever processor), sometimes referred to as “the data plane”.
  3. Local service interfaces used by the worker to obtain various services locally available on the processor on which the worker is running.
- Each authoring model also specifies how a worker is built (compiled, synthesized, linked) and packaged, to be ready for execution in an application.

2.1 Requirements for all authoring models

- Enable/support well-defined data plane interoperability with other authoring models
- Define its OpenCPI Worker Description (OWD) XML format.
- Define programming language interfaces for control, data, and local services.
- Must define the packaging for delivering ready-to-execute workers.

3 Authoring Model Overview

OpenCPI authoring models represent alternative ways of writing code to express the functionality of a *component* that can be used in *component-based applications*. Since a requirement of the authoring models is to achieve the “plug & play” of alternative implementations of the same component functionality, there must be a common way to describe how the component will fit into an application, regardless of the authoring model used by any implementations. This is called the *component specification*.

An *OpenCPI Component Specification (OCS)* describes, in a simple set of one or more XML text files, the aspects of a component that are the same regardless of the authoring model used by any implementation of that component. Thus a requirement for developing an implementation of an OpenCPI component (a *worker*) is that there must exist an OpenCPI Component Specification (OCS) common to all implementations of that component.

The OCS essentially describes two things: (1) the configuration properties of the component (how it is initially and dynamically configured), and (2) the (data) ports of the component (how it talks to other components). Based on these two aspects, all components can be configured and interconnected, regardless of implementation. There is no description of exactly what happens inside the component (a behavioral description), but only how it is used in an application.

[insert example of an OCS here]

The OCS is described in a later section, as a specific format of XML files, and that specific format is used regardless of authoring model. An OCS describes a component's configuration properties, and data ports.

Thus the first step in having actual component implementations built and usable for component-based applications, is to have an OCS on which to base the implementation(s). OpenCPI uses the term “worker” as shorthand for “component implementation”. Thus an OpenCPI worker is an implementation based on an OCS and a particular OpenCPI Authoring Model (OAM). It consists of two things:

1. A separate (form OCS) XML description of the particular implementation, indicating the OAM and which OCS the implementation is based on
2. The source code in some programming language that does the actual computing function of the implementation, written according to the OAM.

The XML description of the worker is called the OpenCPI Worker Description (OWD). Thus to built a worker (to have an implementation) you start with an OCS (common to a group of implementations), write an OWD, and then write the programming language source code.

An OWD is a general concept, since actual OWDs have a format specific to an authoring model. Each authoring model defines a specific XML format for describing the worker written for that model, with the requirement that each OWD refer to the OCS that is common to all workers implementing the same component. The many common aspects of all OWDs are described in the OWD section below.

The following diagram shows the relationships between (multiple) OWDs and an OCS.

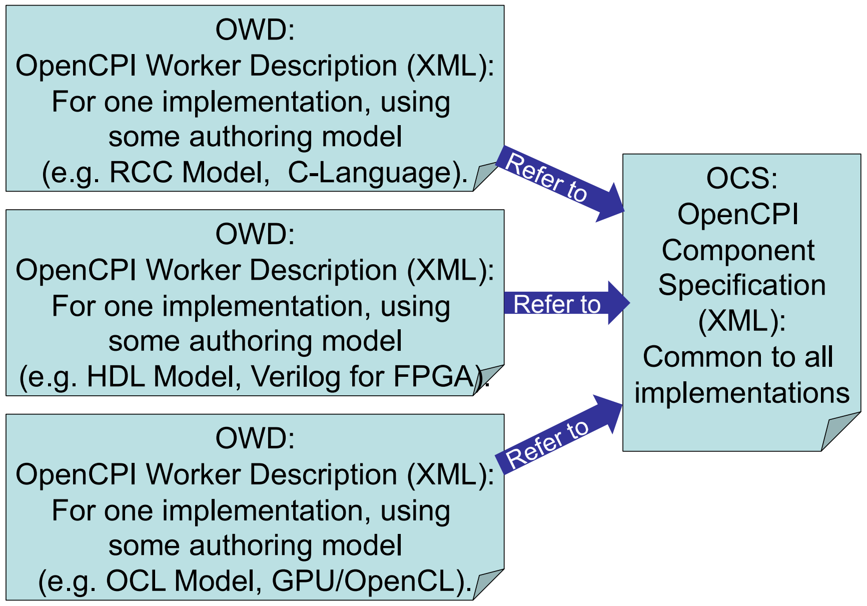
OWD:
OpenCPI Worker Description (XML):
For one implementation, using
some authoring model
(e.g. RCC Model, C-Language).
OWD:
OpenCPI Worker Description (XML):
For one implementation, using
some authoring model
(e.g. HDL Model, Verilog for FPGA).
OWD:
OpenCPI Worker Description (XML):
For one implementation, using
some authoring model
(e.g. OCL Model, GPU/OpenCL).
Refer to
Refer to
Refer to
OCS:
OpenCPI
Component
Specification
(XML):
Common to all
implementations

Figure 1 – Worker Descriptions and Component Specifications

4 Common Aspects to all Authoring Models

4.1 How workers are controlled

OpenCPI uses a "control model" of components that is a subset of that defined by various component system standards such as the DoD's SCA or ISO/OMG CCM. It is intended to be simple and easy to use, while being a proper functional subset so that most OpenCPI workers can be made into compliant components in those systems (for those that care about that compliance). The control model is also designed to be implementable and appropriate across a variety of authoring models and technologies.

Control here refers to configuration mechanisms and a fixed set of *control operations* that every worker, explicitly or implicitly, must support. The control operations are: initialize, start, stop, release, beforeQuery, afterConfigure, finish, and test. Explicit support is when the worker source code itself deals with the control operation. Implicit support is when the OWD indicates that the worker has no code at all for the operation, and thus it has default behavior (described below) implemented or assumed by the infrastructure for the worker. All workers must all be uniformly controllable according to this control scheme.

This control scheme is based on a number of other similar systems, and is intended to be both generally useful and also support enough controllability to be compliant with other component standards. The general model is that workers transition through various states based on the control operations that are issued by "*control software*" that is executing the overall component-based application. It is the responsibility of infrastructure control software to keep track of the states and issue the control operations in the correct sequence. Each authoring model specifies how these control operations are received and acted on by workers. Here is a summary of the control operations, which are detailed below.

Table 2 – Control Operations – Function Overview

Control Operation	Function Overview
Initialize	Perform any post-reset, post-instantiation initializations, before configuration.
Start	Enter the operational state; start doing the operational work.
Stop	Suspend work such that operation can be resumed (via start).
Release	Shut down operations, undoing "initialize".
Finish	Perform final work before application completion.
Test	Run component-level built-in-test, if available.
BeforeQuery	Ensure configuration properties are consistent prior to accessing them.
AfterConfig	Act on multiple interdependent configuration settings.

4.1.1 Control Operation Definitions

In order to keep workers as simple as possible in common cases:

1. It is the responsibility of control software to sequence these operations properly, so workers do not need to check for invalid control operations (issued in the wrong state or in the wrong order)
2. Metadata known to control software, via the OWD, can indicate whether control operations are implemented at all by each worker. See the ControlOperations attribute in the OWD section below.

The operations are:

1. **{Instantiate} (**Implicit by infrastructure/container software/gateware**)**:

- *After worker code loading, the infrastructure creates the runtime instance of the worker. Each authoring model specifies how this actually works.*
- *After this implicit control-operation completes, worker must be ready to accept the **Initialize** operation (if it implements it).*

2. **Initialize:**

- *Needed when additional initialization is required that is not done during instantiation (which has behavior specific to the authoring model).*
- *Allows worker to perform initial work not accomplished during **instantiate**, to achieve a known ready-to-run state*
- *Worker can internally set initial/default values of properties (if not done during **instantiate**)*
- *Worker cannot do one-time work based on software-configured properties. They are not valid yet since control software has not set them yet. Control software will not perform property access until **Initialize** completes.*
- *The OWD for a worker can specify that it has no initialization behavior at all, and thus control software will not bother trying to invoke the non-existent operation.*

3. **Start**:

- *Transition to being operational - to doing the real work based on configuration property values (initialized by worker, and possible set by control software) and input data (at data ports).*
- *Must do any one-time initializations that depend on software-configured property values, since **Initialize** cannot do this (properties are not set yet).*
- *Starting operation is based on property values set after **Initialize** completes, or after worker is suspended by **Stop**.*
- *No accesses should be performed on data ports until this operation is completed by the worker.*
- *Whether this operation is optional (like initialize) or required depends on the authoring model.*

4. **Stop**

- *Make the worker inactive/paused/suspended, such that it can be resumed (via* ***Start****); maintain resources/state/properties for examination.*
- *This is a "graceful" suspension, not an immediate destructive stop or abort.*
- *This operation is optional: when an OWD specifies that it is not supported by the worker, the implication is that the worker cannot be suspended, which may be considered "unfriendly" behavior for application debuggers!*

5. **Release**

- *Shut down/abort operations, discard any state (state is now undefined)*
- *Release any resources obtained in* ***initialize*** *or* ***start****.*
- *If release fails, the worker instance is unusable.*
- *Configuration properties will not be read or written after* ***release*** *completes.*

6. **BeforeQuery**

- *After* ***Initialize****,* ***Start****, or* ***Stop****, informs worker that a group of related property queries (configuration read accesses) will be made and their values should be consistent.*
- *Used to enable coherent access to multiple property values.*
- *Only necessary when property values depend on each other and need to be properly updated to be meaningful/correct as a group.*
- *Some authoring models can omit this operation if configuration properties can be queried in batches (atomically access a group of configuration properties).*

7. **AfterConfig**

- *After* ***Initialize****,* ***Start****, or* ***Stop,*** *informs worker that a group of related property changes have been made.*
- *Informs worker that changes to a set of interrelated property values can be processed.*
- *Some authoring models can omit this operation if configuration properties can be set in batches (atomically set a group of configuration properties).*

8. **Test**

- *After* ***Initialize****, used to run worker-specific built-in tests, parameterized by current configuration property settings.*
- *In a given system or application, the overall system built-in-test can be easily parameterized for each worker by specifying the configuration property settings prior to running "test", and the expected configuration property settings after running "test".*
- *Simple workers that have no such tests can indicate it in OWD.*

4.1.2 Simple workers can be very simple for control operations

For the simplest workers, only the **Start** operation is generally mandatory. The implications are:

1. **Initialize** is not needed or supported– all initialization is performed during instantiation, the worker can accept configuration accesses immediately after instantiation.
2. **Release** is not needed, and the implementation forces control software to reinstantiate the worker instance after being used once. This may reduce system debugging and reloading flexibility, since this "software reset" for re-use is not always possible.
3. **Stop** is not implemented, and thus the worker cannot be paused/suspended at all. This forces whole applications to not be stoppable and thus degrades application and system-level debugging.
4. **Test** is not implemented: there is no built-in test capability.
5. **BeforeQuery** and **AfterConfig** operations are not needed if there are no interdependent property values that must be used atomically/consistently.

Thus simple, but useful, workers are only required to implement the **Start** operation; all others can be specified as unimplemented in metadata. Even the Start operation is optional for some authoring models where it can be implicit.

4.1.3 Control States

Since control software is required to issue control operations correctly, in the appropriate sequence as defined above, these states are not normative, but are used to further clarify the use of control operations.

1. State**: Exists**: after instantiation or successful **release**, ready for **initialize**.

- *The worker is loaded (if necessary), not necessarily fully initialized, configuration properties are not valid, property access is not allowed.*
- *Only the **initialize** operation will be issued (if implemented). If successful the worker is considered to be in the "initialized" state.*
- *If **initialize** is not implemented, worker is **Initialized** after instantiation.*

2. State: **Initialized:** after **initialize** operation**,** ready to **start** (or **release**)

- *The worker has completed any run-time initialization, doing whatever is necessary or appropriate after instantiation.*
- *The worker is in a stable state "ready to start doing work", but not operational.*
- *Configuration properties can be read and written (software can see the result of initialization, and can configure the worker prior to normal operation). Thus the **BeforeQuery** or **AfterConfig** operations maintain this state (don't change it).*

3. State: **Operating:** after **Start** operation**,** ready to **Stop** (or **Release**)

- *The worker is doing normal work based on configuration properties and data at ports; it is "processing".*

- *Configuration properties can be read or written, including* ***BeforeQuery*** *or* ***AfterConfig*** *operations*

4. State: **Suspended:** after **Stop** operation, ready to **Start** (or **Release**)

- *The worker is suspended; worker will not produce or consumer at data ports.*
- *Configuration properties can be read or written, including* ***BeforeQuery*** *or* ***AfterConfig*** *operations*

5. State: **Finished:** after successful ***Finish*** operation, or autonomously from the **Operating** state; ready to **Release**

- *The worker is finished and will not produce or consumer at data interfaces.*
- *Configuration properties can be read or written, including BeforeQuery or AfterConfig operations*

6. State: **Unusable**: after unsuccessful **Release** operation

- *In this state the only action to take is to destroy the worker.*

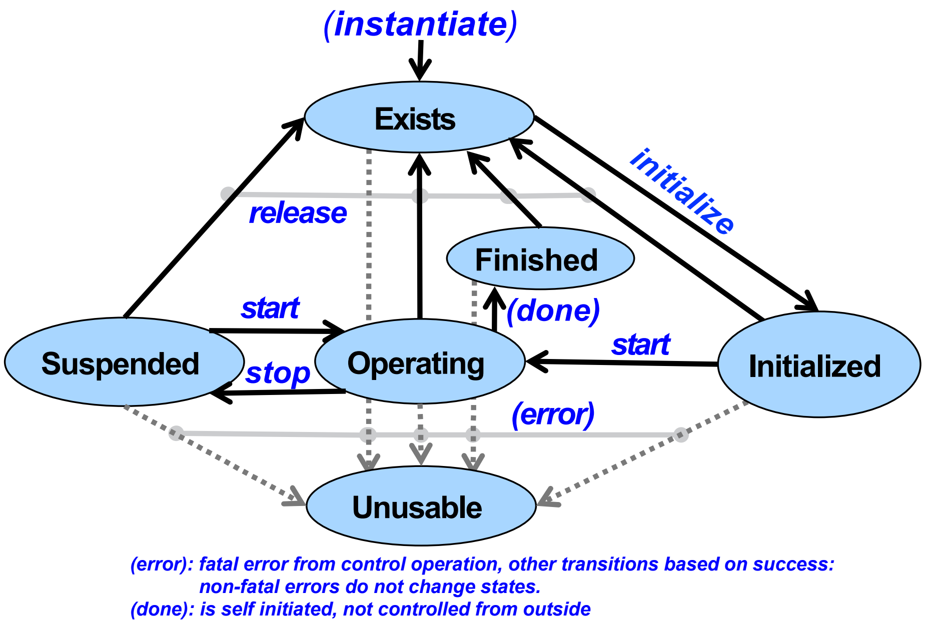
(instantiate)
Exists
initialize
release
Finished
start
(done)
start
Suspended
stop
Operating
Initialized
(error)
Unusable
(error): fatal error from control operation, other transitions based on success: non-fatal errors do not change states.
(done): is self initiated, not controlled from outside

Figure 2 – Worker Control States

4.1.4 Control Operation Errors

Control operation errors are indicated in a way specific to the authoring model.

1. All **release** errors indicate unusable worker state. The worker instance cannot be re-used.
2. If **Initialize**, **Start**, **Stop**, **BeforeQuery** or **AfterConfig** return an error

- *Only Release may be attempted.*

4.2 How Workers are Configured

Configuration refers to the set of, and access to, writeable and readable values, with and without side effects, within the worker. Components may also define their own component-specific specialized control schemes by read and/or write side effects using configuration properties (although this is discouraged). All authoring models must provide the interface enabling control software to access a worker's configuration properties at run time, and also provide access to the worker itself.

Some authoring models define a flat/linear “configuration address space” where the configuration properties are accessed by accessing this memory space, roughly as a data structure or “register file”. This is extremely low latency and high performance, but may not be appropriate for some authoring models. In the case where the configuration property access is via such memory-mapped accesses, the beforeQuery and afterConfig control operations must be defined by the authoring model.

The OCS, as a description that applies to all the implementations of that component specification, contains the description of configuration properties considered part of the “component’s external behavior”: that must exist in all implementations (all workers meeting that spec). However, each implementation may also add to this set of configuration properties and define *implementation-specific* configuration properties. These are sometimes useful for debugging or when the initial configuration specified for the whole application might list some of these properties to properly configure a particular implementation.

Each configuration property is defined with a name and data type from a limited set of data types that are sensible across many authoring models and processing technologies. Although each configuration property may also be variable in length (e.g. a string, or a variable length sequence of numbers), there is always a defined maximum length for any such property value. This enables OpenCPI components and workers to be compliant with a variety of component system standards, and enables authoring models for lean and embedded technologies.

The details of the data types and aggregate types (array, structure, sequence) are defined in the OCS section below. Beyond data type, the configuration properties also have attributes that define whether the value can only be set at initialization time, set any time during execution, or never set (read-only). Similarly, a configuration property value can be described as volatile (the value may spontaneously change during runtime) or static (will not change unless written by control software). This covers many patterns of use and optimizations for such values.

Thus for each configuration property of a worker these aspects are specified:

- *The name of the property*
- *Is it part of the OCS, and apply to all implementations based on the OCS, or is it specific to the OWD — the particular implementation/worker?*
- *What is its data type?*
- *Is it a fixed sized array or variable length sequence?*

- *If it is variable, what is the maximum length?*
- *Can it be changed at runtime, only at initialization or not at all?*
- *Will it change during runtime or is its value constant? or is it only writable, not readable?*
- *If it is an implementation-specific property, is it a build/compiler time parameter?*

Properties listed in the OCS specify the external configuration interface for all implementations of the same spec. From the external/application point of view, the "writable" aspect is either not writable at all, settable at initialization, or settable during execution (after start). The "readable" aspect is either not readable at all, readable for a static/unchanging value, or readable for a "volatile" value that can change during execution. This actually results in 9 valid combinations of readability and writability.

As mentioned earlier, an implementation (OWD) may have additional properties beyond what is specified in the OCS, commonly extra status information for debugging. Furthermore, an OWD may add to the *accessibility* of a property specified in the OCS, e.g. make a property readable that was not readable in the OCS, for debug purposes (e.g. to read back what was written). The accessibility added by the OWD results in the implementation having a superset of what was "promised" in the OCS.

4.2.1 Properties that are in fact build-time parameters

While the OCS specifies properties and their initialization-time and run-time accessibility, an implementation OWD can further declare that a property is actually a compile time parameter of the implementation. This is not allowed (and does not make sense) for properties that are "writable" at runtime. When an implementation has properties that are parameters, it means that the implementation must be built (compiled, synthesized, elaborated) for specific values of such properties. This has three implications:

- *An application can only use the worker if it is built for a property value that matches what is requested as an "initial" property in the application's XML.*
- *Binary component libraries may have multiple builds (binary artifacts) for the same worker, but with different combinations of parameter values.*
- *The component developer must decide which combinations of parameter values to build in order to make alternative settings of such parameters available.*

This parameter feature allows implementations to have compile time optimizations for certain parameter values, and also allow a single worker source code module to be optimized for different values.

Parameter property values are applied to the build process as expected: e.g. by "preprocessor symbol definitions" for software, "generics" for VHDL, and "parameters" for verilog. A very common (and in fact built-in) parameter property is the "debug" boolean property that specifies the typical "debug build" or "production build".

More details about how the various combinations of accessibility interact with parameter properties in the section below defining the details of the OWD.

4.3 How workers communicate with each other

The OpenCPI Data Plane defines how workers communicate with each other. It defines message-oriented, “data-plane” communication. Each authoring model defines how workers receive/consume and send/produce messages to or from other workers, whether collocated in the same device or executing elsewhere (possibly in a different type of device, including other workers implemented in other authoring models).

From the point of view of a worker, it is talking to another worker in the application or some source or sink I/O or network endpoint at the “edge” of the application. The other side of the data plane interface may be infrastructure software/gateware in these cases:

- The other worker is local (running in the same execution environment on the same processor), but needs an adapter for the two data interfaces to interoperate.
- This worker is at the “edge” of the application, and the data is being sent to or received from some external source, file, or I/O device. Example: data is being received from an A/D converter, and the worker is receiving the output of the A/D via the infrastructure software/gateware that deals with the A/D hardware (its driver or “*device worker*”).
- The other worker is remote, in another device, requiring infrastructure to forward messages across some interconnect or fabric. Example: a worker is producing data for a consuming worker in a different processor connected by a fabric or bus attached to the local processor or FPGA.

4.3.1 Data Protocol between Workers: the Data Plane Protocol

Each worker data port conveys data content between workers according to a data protocol described by metadata attributes. The definition and format of actual message payloads are defined in (or referenced by) the OCS. Thus each worker implements the data ports/interfaces specified in the OCS and consumes or produces messages of the type and payload defined by each port's message protocol defined by or referred to in the OCS.

The simplest case of the “protocol” is a sequence of single-data-value messages of a single message type. This is the way to describe such a continuous stream of data values:

- There is one type of message.
- The message length is fixed as the size of a single data value (and thus no need for an explicit message length, e.g. if the data values are single precision IEEE floats, all messages are 32 bits/4 bytes).

A protocol is specified for each of the worker’s data ports, independent of authoring model, or other implementation-specific attributes. While the OCS specifies the protocol as a set of possible message types and payloads, each authoring model specifies how a worker sends and received messages at each of its ports. The authoring model must also specify how a worker knows whether a port is actually connected to another worker or not (when the OCS for that port specifies that it may in fact be unconnected in the application).

A protocol may define messages in both directions, thus a worker's ports are either unidirectional (when the protocol only includes messages in one direction) or bidirectional (when the protocol includes messages in both directions). The roles in the protocol when it is unidirectional can simply be thought of as a producer and a consumer. When bidirectional, the protocol is analogous to "client or user" vs. "server or provider". In OpenCPI, the bidirectional protocol concept is a rough subset of the "request" and "reply" semantics of CORBA or other RPC systems. Authoring models may require that, on a given port, replies are always produced by the "server" in the order that requests were received. [Note bidirectional ports are a roadmap item].

5 The OpenCPI Component Specification (OCS)

This section defines the XML file format for creating OpenCPI Component Specification XML documents that describe the implementation-independent specification as the basis for one or more worker implementations that might use different authoring models. This concrete document format is referenced from implementation-specific description documents (OWDs) that are defined in each authoring model.

5.1 Quick XML Introduction

XML documents are text files that contain information formatted according to XML (Extensible Markup Language) syntax. XML information is formatted into Elements, Attributes, and Textual Content. OpenCPI XML files do not use Textual Content at this time. XML Elements take two forms. The first is for when an element has no child (embedded) element. It looks like this (for element of type , “xyz”, with attribute “abc”):

```
<xyz abc=”123”/>
```

The element begins with the “<” character and the element type, and ends with “/>”. Attributes values are in single or double quotes. White space, indentation, or new lines can be inserted between the element name and attributes. So the example could be:

```
<xyz
   abc=”123”
   />
```

When the element has child elements (e.g. element “ccc” with attribute “cat”), it is:

```
<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 s*chema* defines which Elements, Attributes, and child Elements the document may contain. Every XML document (in this context) has a single top-level element that must be structured (attributes and sub-elements) according to the OCS XML schema.

An element can be entered directly (as above) or entered by referring to a separate file containing the 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>
```

The XML schema specifies which elements are allowed at the top-level in any file. The default value of all boolean attributes is false unless otherwise noted. At this time no explicit XML schema for OpenCPI is available, as no tools require it.

When XML files are processed in OpenCPI tools, any internal file references (where one XML file refers to another), are subject to directory search paths and rarely use absolute or relative pathnames. Either the referenced file is in the same directory as the first file, or it is in a directory that is in the search path.

5.2 Top Level Element: OpenCPI Component Specification

This top-level XML element is the basis for the OCS.

A component specification is the XML element describing a component at a high level *without implementation choices or details*. It describes enough about how the component can be integrated into an application to ensure interoperability and interchangeability among different implementations. Multiple such implementations are possible given the *component specification*. A component specification is sometimes derived from a higher-level component model such as that defined in the various software-defined radio component standards such as SCA and the OMG's "PIM/PSM for software-based communication", as well as the generic component standards such as the ones in UML v2 or CCM v4.

The component specification element contains component-global attributes, control plane aspects and data plane aspects. A component specification is contained in the XML element whose type is "ComponentSpec", which should be a top-level element in a file, structured as:

```
<ComponentSpec

   ---attributes---
      >
   ---child elements---
</ComponentSpec>
```

5.3 Attributes of a Component Specification

5.3.1 Name

The “Name” attribute of the component specification is constrained to be acceptable as an identifier in several contexts, including various case insensitive programming languages. It identifies the component specification as a whole. It is case *insensitive*: when in a library or application, two different component specifications cannot differ only in case.

5.3.2 NoControl

The “NoControl” attribute of the component specification is a boolean attribute that indicates, when true, that components using this specification have no control plane/interface at all. This is not allowed for application components but is specified for certain infrastructure components.

5.4 Control Plane Aspects of a Component Specification

A component's control plane specification is essentially a description of its configuration properties used to parameterize its operation or to retrieve operational statistics and resulting scalar values based on the component's operation. The Properties element of a component specification enumerates the name and type and behavior of each configuration property supported by implementations of this OCS.

The Properties element may be in a separate file and referenced using the <xi:include href="<file>"/> syntax. This is common when groups of Property elements are shared among multiple component specifications.

5.4.1 Properties Element

The Properties element has no attributes, but consists of a list of Property elements. Anywhere in the list of Property elements in the overall Properties element, you can be include additional Properties elements using the <xi:include href="<file>"/> syntax. This allows a list of properties to be shared as a subset of the properties of different components. When a group of properties is used in multiple component specs, it is placed in a separate file with this element at the top level. When the properties are specified directly under a "ComponentSpec" element, the Property element can be used directly without a Properties element to contain them.

5.4.2 Property Element

A Property element describes one configuration property. It occurs as a sub-element of either the ComponentSpec element or the Properties element. A Property element describes the name and data type of a configuration property, and there is a "struct" data type to allow properties to consist of a structure of data members. There is no support for recursive structures. Each property can also be an array or sequence of its data type. Consistent with the CORBA IDL specification, the term "array" refers to a fixed number of data values of the specified type, whereas the term "sequence" refers to a variable number of data values, up to a specified maximum length. The attributes of the Property Element are listed here:

5.4.2.1 Name attribute

The Name attribute is the case insensitive name of the property. A set of properties cannot have properties whose names differ only in case. Mixed case property names can be used for readability. When a Properties element includes (via xi:include) other Properties elements there is still only one flat case-insensitive name space of properties for the component.

5.4.2.2 Type attribute

The Type property specifies the data type of the property. The legal types (case insensitive) are: Bool, Char, Double, Float, Short, Long, UChar, ULong, UShort, LongLong, ULongLong, Enum, and String. The "Char", "Short", "Long", and "LongLong" types represent 8, 16, 32, and 64 signed integer values respectively. The "Float" and "Double" types are consistent with 32 and 64 bit IEEE floating-point types, and the

“String” type is a null terminated string. When the “Type” attribute has the “String” value, the “StringLength” attribute must also be supplied, to indicate the maximum length of the string property values, excluding any terminating null character (consistent with the ISO-C strlen function). If no Type attribute is present in the Property element, the type “ULong” is used.

When the type is “Enum”, the actual values are Ulong, but the values are indicated by strings found in the “enums” attribute described below.

The “ArrayLength” attribute is used when the property is a fixed length one dimensional array of the indicated type. The “SequenceLength” attribute is used when the property is a variable length sequence of the indicated type.

When the type is “Struct”, the Property element has Property sub-elements that indicate the types of the members of the struct value. No struct members can be of type “Struct”. The SequenceLength and ArrayLength attributes may apply to “Struct” properties.

Thus all types have a maximum length. Properties cannot have unbounded length.

5.4.2.3 StringLength attribute

The StringLength attribute is used when the Type attribute is “String”, and indicates the maximum length (excluding any null termination) string value that this property can hold.

5.4.2.4 Enums attribute

This attribute is required when the Type attribute is “enum”, and its value is a comma-separated list of strings that name the enumerated values. The actual values are Ulong and are zero-based ordinals based on the position of the names in the list.

5.4.2.5 ArrayLength attribute

The presence of this attribute indicates that the property values are a fixed length one-dimensional array of the type specified in the Type attribute, and that fixed length is indicated in the value of this attribute.

5.4.2.6 SequenceLength attribute

The presence of this attribute indicates that the number of property values is a variable, but bounded, sequence of the type specified in the Type attribute, and that maximum length is indicated in the value of this SequenceLength attribute. Thus this property has the specified maximum length, and always contains a current length, up to that limit. When both SequenceLength and ArrayLength attributes are present, the meaning is “sequence of arrays”, *not* “array of sequences”.

5.4.2.7 ArrayDimensions attribute

The value of this attribute is a comma-separated list of array dimensions indicating an array whose number of dimensions is the number of values in the list. If this attribute is set, then the ArrayLength attribute should not be set. This means values are multidimensional arrays with elements whose type is indicated by the Type attribute.

5.4.2.8 “Default” attribute

This string attribute provides a default value for the property for all implementations. It is parsed based on the data type specified in the Type attribute. This value is set by the infrastructure at runtime when any implementation is instantiated in the runtime environment, unless an initial property value is specified at the time the worker is created. The purpose of this attribute is to advertise what value will be used if no initial value is provided by the application. Default values should only be specified for properties with that are “initial” or “writable” (see below).

5.4.3 Accessibility Attributes of Property elements

The attributes described here specify what property accesses are allowed under what conditions. They are all boolean attributes and all have the default value of “false”. It is an error for all of them to be false.

5.4.3.1 Readable attribute

This attribute indicates whether this property can be *read* by control software. If set to false (as well as the “volatile” attribute being false), attempts to read the property value at any time after instantiation will result in an error.

5.4.3.2 Volatile attribute

This attribute indicates, when true, that the property is readable *and* that its value may change during execution, without it being written by control software. When neither “readable” nor “volatile” is true, attempts to read the property value will result in an error. Only one of “readable” and “volatile” should be set to “true”.

5.4.3.3 Writable attribute

This attribute indicates whether this property can be written either during and after initialization by control software. If set to false, attempts to write the property value at any time *after* initialization will result in an error.

5.4.3.4 Initial attribute

This attribute indicates whether this property can be set at the time of instantiation as an *initial* value, by control software. If set to false (and “writable” being false), attempts to specify the property at instantiation time will result in an error. Thus a property that has “Initial” as true, but writable as false, can be set at initialization-time, but not set at run-time. Only one of “writable” and “initial” may be set to “true”.

5.4.3.5 Padding attribute

This attribute is used for properties that exist only for “padding” purposes, and thus are otherwise inaccessible. In some cases where the list of properties is intended to be in a predetermined sequence with specific offsets or gaps between properties, a property that has no access is used, and needs this attribute to be set to “true”.

5.5 Data Plane Aspects of a Component Specification

5.5.1 Data Interface Specification Element

The component specification defines data plane ports through the use of the "DataInterfaceSpec" element. It specifies the direction/role of the interface (producer or consumer) and the message-level protocol used at that interface.

The DataInterfaceSpec element has several attributes and one child element: the Protocol.

5.5.1.1 Name attribute

This attribute specifies the name of the interface for this component. The value of the name attribute is a string that is constrained to be valid in various programming languages. It must be unique (case insensitive) within the component specification.

5.5.1.2 Producer attribute

This boolean attribute indicates whether this data interface has the role of a producer/client/user (when "true"), vs. the default (false) of consumer/server/provider.

5.5.1.3 Protocol attribute

Not be confused with the Protocol ***element*** below, this string attribute names an XML file containing the protocol description for this port. The named file is expected to contain a Protocol ***element*** at its top level.

5.5.1.4 Optional attribute

This boolean attribute indicates whether this data port may be left unconnected in an application. The default (false) indicates that workers implementing this component require that this port have a connection to some other worker in the application. When true, this data port may be left unconnected and all workers implementing this specification must allow for the case when the port is not connected to anything.

5.5.2 Protocol element

The protocol element, which is a child element of the data interface specification element, specifies the message protocol used at this port. The Protocol is a separate element since it will likely be reused across a variety of components and interfaces and thus may be referred to in a separate common file via the "xi:include" mechanism mentioned above or more simply using the "protocol" attribute described above. The protocol (element) has "Operation" subelements to indicate the different message types that may flow over this data port.

If the port being described is permissive, meaning it can accept any protocol, then this element can be absent. An example of this is a "file writing" component that accepts any types of messages as input, regardless of protocol.

5.5.3 Operation subelement of the Protocol element

The term “Operation” is loosely associated with the analogous concept in RPC systems were the message is “invoking an operation on a remote object”, but in this context it simply describes one of the messages that is legal to send on a port with this protocol. It has two attributes and some number of “Argument” subelements.

5.5.3.1 Name attribute of Operation element

The type of this attribute is of type string, and is the case insensitive name of the operation/message within this protocol.

5.5.3.2 TwoWay attribute of Operation element

This boolean attribute indicates whether this operation is a two-way operation with outbound “request” messages from producer/user/client to consumer/provider/server, as well as a corresponding “reply” messages back from the consumer/provider/server to the producer/user/client. The default is false (as with all boolean attributes in this specification). [twoway support is a roadmap item]

5.5.4 Argument subelement of Operation element

The subelement indicates a data value in the message payload for the given “operation” (message type). Its attributes are the same as those of the “Property” element that describes a configuration property (Name, Type, StringLength, ArrayLength, SequenceLength, etc.). If no argument elements are present it means that the operation carries messages with no data (sometimes called Zero Length Messages). Different than properties, arguments to an operation do *not* have to have bounded lengths, so the StringLength attribute is not required for strings, and the SequenceLength attribute is not required for sequences.

5.6 Component Specification Examples

```
<ComponentSpec Name="K1spec"
   <Properties>
      <Property Name=”size” Type=”float”/>
   </Properties>
   <DataInterfaceSpec Name="lvds_tx" Producer="true">
      <Protocol>
         <Operation Name=”mess1”>
            <Argument Name=”val” Type=”uShort”
               ArrayLength=”1024”/>
            </Argument>
         </Operation>
      </Protocol>
   </DataInterfaceSpec>
</ComponentSpec>
```

6 The OpenCPI Worker Description (OWD)

The OpenCPI Worker Description (OWD) is not completely specified here, since it is customized for each authoring more. Actual OWDs are defined for each authoring model in separate documents, and the top-level element is named according to the authoring model. For example, the RCC authoring model defines an OWD where the top-level element in the OWD XML file is “RCCWorker”. This section describes aspects common to the OWDs for *all* authoring models. (For XML fans, it is roughly an inherited schema).

OWDs for an authoring model called XYZ have a top-level element called XYZWorker. This top level element must refer to an OCS by:

- containing a ComponentSpec child element (not shared with any other OWD)
- indicating an OCS file by using the “spec” attribute.
- contain a reference to a ComponentSpec via “<xi:include href=”filename”/>

Thus the OWD says which OCS it is implementing via one of these mechanisms

An OWD contains information about an implementation that is based on a component specification. It includes or references a component specification, and then describes implementation information about a particular implementation of that specification. For example, if the “fastcore” implementation of the “corespec1” specification referenced the component specification found in the “corespec1.xml” file:

```
<HdlWorker  Name=”fastcore” Spec='core1-spec.xml'
   ---other attributes---
   >
   ---other child elements---
</HdlWorker>
```

6.1 Attributes of an OWD

While an OWD may well have attributes specific to its authoring model, the common ones are defined here. Attributes that are specific to an authoring model are *not* defined here, but there are a few implementation-specific attributes that may occur in several OWDs so those are defined here also.

6.1.1 Name attribute of the XYZWorker element.

This top-level element has a "Name" attribute which defaults to the name of the OWD XML file itself (without directory or extension). The "Name" attribute of the component implementation is constrained to be acceptable as an identifier in several contexts, including case insensitive languages. It is sometimes called the "worker name" or "implementation name".

The name of the implementation may be the same as the name of the OCS (the "spec name"). It is typically not required to have a unique name for the OWD unless there are multiple implementations of one OCS that use the same authoring model. I.e. OWD names are implicitly scoped by authoring model.

6.2 Control Plane Aspects of an OWD

6.2.1 Spec attribute of the XYZWorker element

This string attribute specifies the name of the file for the OCS for this worker.

6.2.2 Language attribute of the XYZWorker element

This string attribute specifies the source code language used in this worker. The valid languages depends on the authoring model, and for each model there is a default language. Some authoring models have only one valid language in which case this attribute is not used.

6.2.3 ControlOperations attribute of the XYZWorker element.

This attribute contains a comma-separated list of strings identifying the implemented control operations. For operations that are mandatory for the authoring model, they are assumed. Thus the default implies a minimal implementation that only implements those operations required by the authoring model. The control operations are listed above. Control operations that are required by the authoring model do not need to be mentioned. When only mandatory operations are implemented, this attribute need not be mentioned.

6.2.4 Property and SpecProperty subelements

Whereas an OCS can only contain “Property” sub elements, an OWD can contain both Property and SpecProperty elements. The “Property” elements introduce new implementation-specific properties unrelated to those defined in the OCS. The “SpecProperty” elements *add* implementation-specific attributes to the properties already defined in the OCS.

Implementation-specific property *attributes* can occur either in the Property or SpecProperty elements here. Property elements in the OWD support all the attributes for Property elements in the OCS (described above) as well as any implementation-specific attributes. For attributes that apply to the Property elements, any specific contraints for OWDs vs. OCSs are noted below.

6.2.4.1 Name attribute for OWD Property or SpecProperty elements

The Name attribute is the case insensitive name of the property. The “Name” attribute is used in *SpecProperty* elements to indicate which OCS property is being referenced. In the *Property* elements it indicates the name of the implementation-specific property, which must not be the same as any Property element in the OCS.

6.2.4.2 ReadSync and WriteSync for OWD Property or SpecProperty elements

These boolean attributes (default false) are used to indicate those properties that require the use of the beforeQuery and afterConfig control operations. I.e., they indicate the need of the worker to know when a group of property settings which include the indicated properties is about to be read (ReadSync) and thus first require a beforeQuery control operation, or that such a group has been written (WriteSync) and thus require an afterConfig control operation after the group is written. Authoring models that have some native mechanism for reading or writing properties in atomic “batches”, have no need for these attributes or control operations.

6.2.4.3 ReadError/WriteError attributes for OWD Property/SpecProperty elements

These Boolean attributes (default is false) indicate properties that may return errors when the property is read (ReadError) or written (WriteError). If a worker does no such error checking and always succeeds when such property values are read or written, then leaving these values false allows control software to avoid any error checking, which in some models and systems can carry significant overhead. Many workers in most models simply accept new values and thus the default is false.

Some authoring models may always convey the error indications as part of how property values are read or written, in which case these attributes are not necessary.

6.2.4.4 “Default” attribute for OWD Property/SpecProperty elements

This string attribute provides a default value for the property for this implementation. If the Default attribute is specified in a SpecProperty element, it is providing a default value for this implementation only. It is not permitted to provide a default value in a SpecProperty when the property in the OCS already has a default value. This attribute's value is applied consistent with the access attributes, specifically:

- For an “initial” or “writable” property, the value is set into the implementation at initialization, if no other initial value is specified in the application.
- For a “readable” property, the value will be statically available to control software. If it is also writable, the value may be overridden at run-time after initialization.

6.2.4.5 Parameter attribute

This boolean attribute, in a SpecProperty element or a Property element, indicates that the property's value is used at compile/build time when the source code is processed into a binary artifact to be loaded and executed and run-time. Parameter properties are supplied to the compilation process for the worker, in a form appropriate for the language and authoring model used. The actual values supplied, use the default value attribute (described just above), except when overriden by values specified in the build directory for the worker (as described in the Component Developer Guide). Thus the OWD just specifies that the property is a build parameter, and can supply a default value for the parameter.

Parameter properties are utilized for other purposes depending on the access attributes and whether the property is in the OCS or only in the OWD.

- When a parameter property is “readable” or “volatile”, the value supplied to the build process is also available as a readable property at runtime, with a constant value.
- When a parameter property is “initial”, the binary worker is only considered usable for the application if the value supplied to the build process matches the value specified (required) by the application.

7 Glossary

Application – In this context of Component-Based Development (CBD), an application is a composition or assembly of components that as a whole perform some useful function. The assembly specifies initial property values for each component. The term "application" can also be an adjective to distinguish functions or code from "infrastructure" to support the execution of component-based application. I.e. software/gateware is either "application" or "infrastructure".

Configuration Properties – Named values of a component or worker that may be read or written by control software. 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. Configuration properties with side effects can be used for custom worker control. Each worker may have its own, possibly unique, set of configuration properties.

Control Operations – A fixed set of controlling operations that every worker has. The control aspect is a common control model that allows all workers to be managed without having to customize any software for each worker, while the aforementioned configuration properties are used to specialize components.

Infrastructure – Software/gateware is either application of infrastructure.

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

Authoring Model – A set of metadata and language rules and interfaces for writing a worker.