OpenCPI Technical Summary

Authors: Jim Kulp

Revision History

Revision	Description of Change	By	Date
1.01	Creation	jkulp	2010-5-01

Table of Contents

1 Introduction

1.1 References

This document is intended to be standalone and not have other document prerequisites, although some familiarity with XML is useful.

Table 1 - Table of Reference Documents

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

1.2 Purpose

The purpose of this document is to provide an overview of OpenCPI and its technology and architecture, and provide common introductory material for other OpenCPI documents. It will provide prerequisite information for other documents, and be a common starting point to understanding what OpenCPI actually is and does.

The key attributes of OpenCPI as presently constituted are:

- Infrastructure, in software and gateware, to support component-based applications on heterogeneous embedded systems.
- Portability and runtime interoperability among components authored for FPGAs, DSPs, GPPs, (and GPUs — in progress).
- Support for using X-Midas-based algorithms (Intel Community SIGINT toolkit)
- Easily adapted to support high-level component-based environments such as DoD SCA/SDR and ISO CCM.
- Easily adapted to support a wide range of data transport technologies ranging from simple wires between chips to high-level middlewares such as DDS and CORBA.
- Architected for high bandwidth, high utilization embedded applications, including CREW/CIED, EO/IR Sensor processing, SATCOM, etc.

These attributes produce the following benefits for embedded systems developers:

- Technology agility to select, change, upgrade, mix processing technologies to stay with best of breed technologies, meet SWAP, and fit the technology to the application(s).
- Diversity of component/algorithm authoring models to exploit existing external modules (e.g. FPGA or X-Midas primitives), and anticipate technology migration of algorithms.
- Improve risk and time-to-deployment when integrating capabilities developed by different organizations, or reused from previous deployments.

2 Overview

OpenCPI is an open source software (OSS) framework to simplify complexity and improve code portability of embedded/real-time systems. OpenCPI is a middleware solution that simplifies programming of heterogeneous processing environments consisting of field-programmable gate arrays (FPGA), general-purpose processors (GPP), digital signal processors (DSP), and a variety of interconnection technologies including high-speed switch fabrics.

Building on some of the concepts introduced by the U.S. Government's Software Communications Architecture (SCA) standard, as well as other component-based frameworks and standards, OpenCPI extends component-based architectures into FPGAs, DSPs and GPUs to decrease development costs and time to market through code portability, reuse, and ease of integration. OpenCPI is based on about 30 man-years of funding.

OpenCPI is targeted toward real-time signal processing for embedded, heterogeneous systems for communications, xxxINT (Signals, Communications, Electronics, etc. - Intelligence) and CIED (Counter Improvised Explosive Devices) in defense intelligence, surveillance and reconnaissance (ISR) systems.

2.1 Component-based development/architecture/systems

The concepts of component-based development (CBD) and component-based architectures (CBA) have been used for more than a decade, and were developed in the 1990s as an evolution of object-oriented concepts. However, its application to embedded systems in general, and heterogeneous multi-processor embedded systems in particular has been minimal. The basic concept is that the "target platform" for the application is a collection of processing resources (nodes, chips) attached to some common interconnect technologies (busses, wires, fabrics, networks). Such platforms can in some sense be thought of as a "distributed system" in that there are multiple processors connected on a "network", but in heterogeneous multiprocessing configurations for embedded systems, the "network" is typically itself a heterogeneous collection of in-the-box interconnect technologies such as PCI Express, RapidIO, EMIF, backplane Ethernet, etc.

So OpenCPI takes the concepts of CBD/CBA and provides an environment and framework to use those concepts in this class of embedded systems. "Using the concepts" means developing and running component-based applications.

The concept of a component-based "system" is summarized in the following diagram, where applications consisting of components are "deployed" by placing components into "containers" which are execution environments running on computing nodes. The communications between the components of an application must be implemented across a variety of interconnect technologies, including the special case where the two components are in fact in the same container. The hardware platform is computing nodes, interconnect nodes, and I/O devices. The application is a set of communicating components.

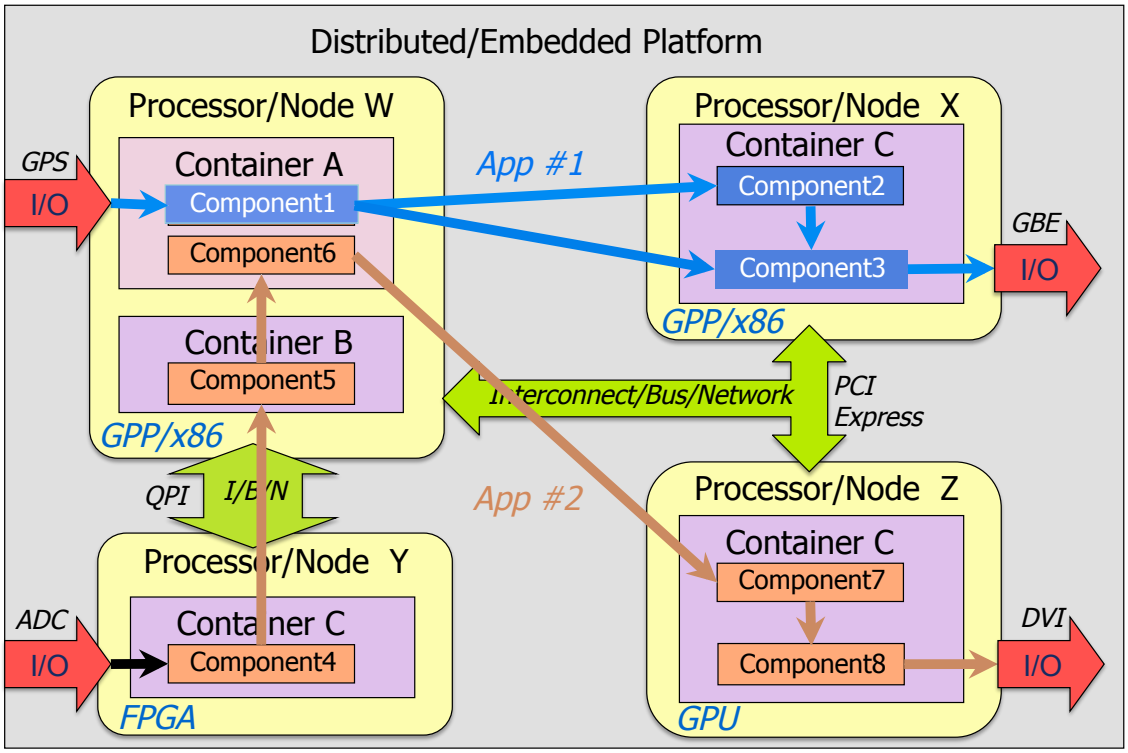

Figure 1: Component-based Systems and Applications

3 What is a component?

A ***component*** is a building block for applications that runs inside a "***container***", which is an execution environment that may be a process on a GPP, or a platform set of gates on an FPGA, or any context that supports the component in performing its function. The concept encompasses functional units implemented in software or hardware.

Components are modules which offers functionality through explicit interfaces, usually called "***ports***", and are reusable parts that have a physical packaging of alternative implementation elements: perhaps compiled binaries for different processors and/or operating systems. Components are defined to be independently deliverable packages of functionality that can be used to build (or "compose") applications or larger components. They are units that are pre-built, packaged, self-describing, which can be individually deployed or updated or replaced in the field. They can be sent as email attachments and patched into installed applications.

Fundamentally they are building blocks for applications, having a functional and interface contract, which includes configuration properties, and ports (for talking to other components), that use or provide data or services and have defined protocols/interfaces, such as:

- Streaming (Media, DRI, Sensor, etc.)
- Request/response (RMI, CORBA etc.)
- Publish/subscribe (Events, DDS, etc.)

Components are "active" (autonomous, concurrently executing), with their behavior usually based on interactions via their ports, with other components. So components are "*specified*" according to their configuration properties (runtime scalar parameters), and their ports (that use certain protocols), and then are "*implemented*" for different execution environments (containers) and technologies (processors). The component implementations are written in a variety of "***authoring models***": languages and APIs convenient and appropriate for both algorithms and target processors. OpenCPI allows applications to be composed from component implementations ("***workers***") written to a variety of ***authoring models***.

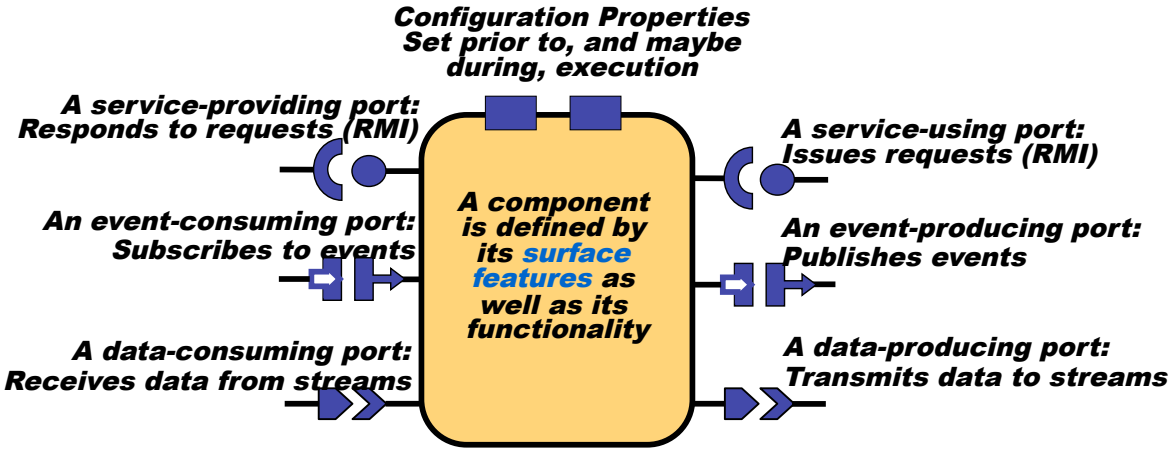

Figure 2: Component Abstraction

4 Authoring Models

We define an OpenCPI ***authoring model*** casually as “a way to write a worker”. In the OpenCPI component framework a primary goal is to support (i.e. utilize for computing) different processing technologies such as General Purpose Processors (GPPs: like Intel/AMD x86 or Freescale PPC), Field-Programmable Gate Arrays (FPGAs: like Xilinx or Altera), Digital Signal Processors (DSPs: like Texas Instruments or Analog Devices), or Graphics Processors (GPUs: like AMD/ATI or NVidia).

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 OpenCPI authoring 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 model and technology used for a particular component in a component-based OpenCPI application, without affecting the models or processor technology used for 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.
- Three sets of interfaces used for interactions between the worker itself and its environment:
  1. Control and configuration interfaces used to control the lifecycle and configuration of workers in the runtime environment, 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.
- The authoring model also specifies the method for how a worker is built (compiled/synthesized/linked) to be ready for execution in an application, given a proper OpenCPI tools implementation for that authoring model.

4.1 Requirements for all authoring models

- Well-defined interoperability with other authoring models (they can coexist)
- Must support compliance (as a subset) with other more complex and standardized component systems such as DoD SCA and ISO CCM.
- Must define the metadata schema for implementation choices.
- Must define programming language interfaces for control, data, and local

services.

- Must define the build flow for creating ready-to-execute workers.

The pattern used for an authoring model is depicted in the following diagram. The three elements of the pattern are:

- The Management Interface for configuration and lifecycle control, termed the "control plane".
- The interfaces and APIs for communicating (via messages) to other components (written in *any* authoring model).
- The local services (sometimes termed the AEP: Application Environment Profile) that the worker can use to perform its function.

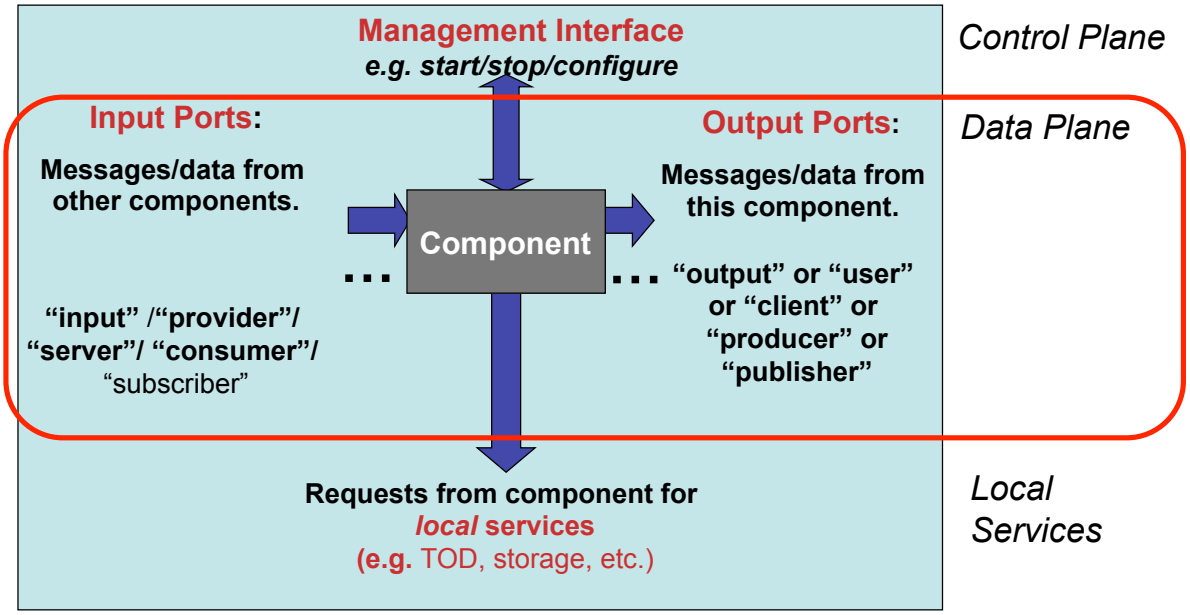

Figure 3: The Authoring Model Pattern

At the time of this writing the OpenCPI authoring models include:

- RCC (Resource Constrained C-language) for using the C language on software systems ranging down to microcontrollers, DSPs, dedicated cores of a multi-core, etc.
- HDL (Hardware Description Language) for components executing on FPGAs.
- XM (X-Midas) for wrapping X-Midas primitives to function as OpenCPI components
- OCL (OpenCL-based GPU targeted): for components executing on and written for graphics processors (GPGPUs) (this work is in progress).
- SCA components (using C++, POSIX, and CORBA) written to be compliant with the DoD's software-defined radio (SDR) standard.

The key concepts of authoring models is the balance between being interoperable (so they can be mixed within a system or application), replaceable (so a part of an application can be changed from one to the other), and efficient (low overhead).

5 Software Architecture

At the highest level, the architecture of OpenCPI (the core software and gateware) is triangulating between three key concepts:

Management Model: How are component-based applications managed and controlled including loading, launching, starting, stopping, configuring, querying, etc. This is sometimes described as how component-based applications are "deployed".

Authoring Model: How components are written in order to be effective on various processing technologies and execution environments.

Data Transport: How messages are moved between one component and another (or multicast etc.).

The core software is thus structured to allow extensions in any of these three dimensions: enable new management models, add new authoring models, and adding new data transport technologies. These three primary dimensions of adaptation are shown in the diagram below.

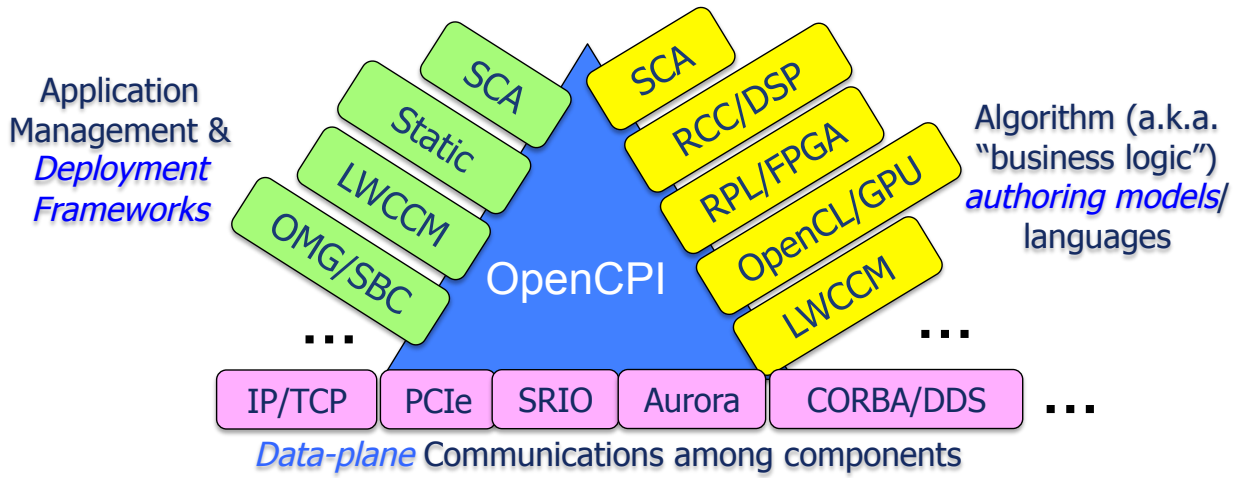

Figure 3: Three Dimensions of Adaption

5.1 Management Models

There are a variety of component based application standards that specify how applications are managed and deployed, including how a "control application" that is not component based, might statically or dynamically decide to execute one or more component-based applications on a given system.

One key such standard is the DoD SCA (Software Communications Architecture) that was specified under DoD supervision, and was targeted at Software Defined Radio (SDR) applications, usually termed "waveforms". Another, called CCM (for the CORBA Component Model) was defined and established in the OMG (Object Management Group), the standards body in charge of the CORBA and UML standards, as well as with ISO (the International Standards Organization). There are several others.

OpenCPI has its own lean and simple native C++ API for controlling and launching applications, but it also is architected to be adapted to these other standards-based

environments (which are necessarily more complex). Thus an SCA-compliant system can be built from OpenCPI by adding an adaptation layer that allows the workers and containers supported by OpenCPI to be managed in an SCA-compliant fashion.

As seen in the diagram above, the SCA defines both a management model as well as a component authoring model. OpenCPI supports them both, but does so in a way that keeps them independent. SCA-compliant components can exist (be executed) in an environment that is not necessarily managed by an SCA-compliant management system (called, in SCA parlance, a "Core Framework"). Conversely, an SCA management framework can be used even if none of the components being executed were written to the SCA component authoring model.

5.2 Authoring Models

The authoring models were discussed above. The architecture of the OpenCPI software base is structured to easily add support for the runtime support for new authoring models. This is based on a "container" plug-in model where container software (and gateware) can easily be configured into the system based on the need for supporting specific authoring models on specific processing devices. This can be done either using static or dynamic linking. Similarly, the workflow and tools that support the development of component implementations (workers), under a variety of authoring models is extensible to allow adding the tools aspects of authoring models along with the runtime (container) support for them.

5.3 Data Plane Support

The "data plane" of OpenCPI is the software and gateware that enables components to talk to each other. The communication model of OpenCPI is conceptually based on a "protocol" model where a component is defined to be able to send or receive messages with defined payloads, and thus is roughly based on the "IDL" concept (Interface Definition Language) where a metadata description of a message-based protocol is established separately from defining components, and then a component specification declares that a given port supports a given protocol consisting of messages and payloads. The "protocol" can be as simple is "frames of 200 16 bit unsigned integers", to a variety of messages in a variety of formats based on variable length data types.

The software architecture of OpenCPI allows this "message transport" to be accomplished using a wide variety of data transport technologies, including:

- passing buffer pointers with no data copying between collocated workers in software
- directly connecting (gutlessly) wires from one FPGA worker to another
- moving messages over network sockets (TCP/IP)
- moving messages over switched fabrics with RDMA hardware
- moving messages over higher level middleware such as CORBA or DDS
- moving messages between processors using dedicated wires between chips.

Any of these choices is accomplished by a "data transport driver" plug-in architecture at a very low level in order to efficiently use low-level hardware while also supporting high level middleware.

5.4 A layered view

While OpenCPI restricts itself to a few key layers (containers, data transports, timekeeping etc.), it is designed to fit into a notional stack of software and gateware, as shown in this diagram:

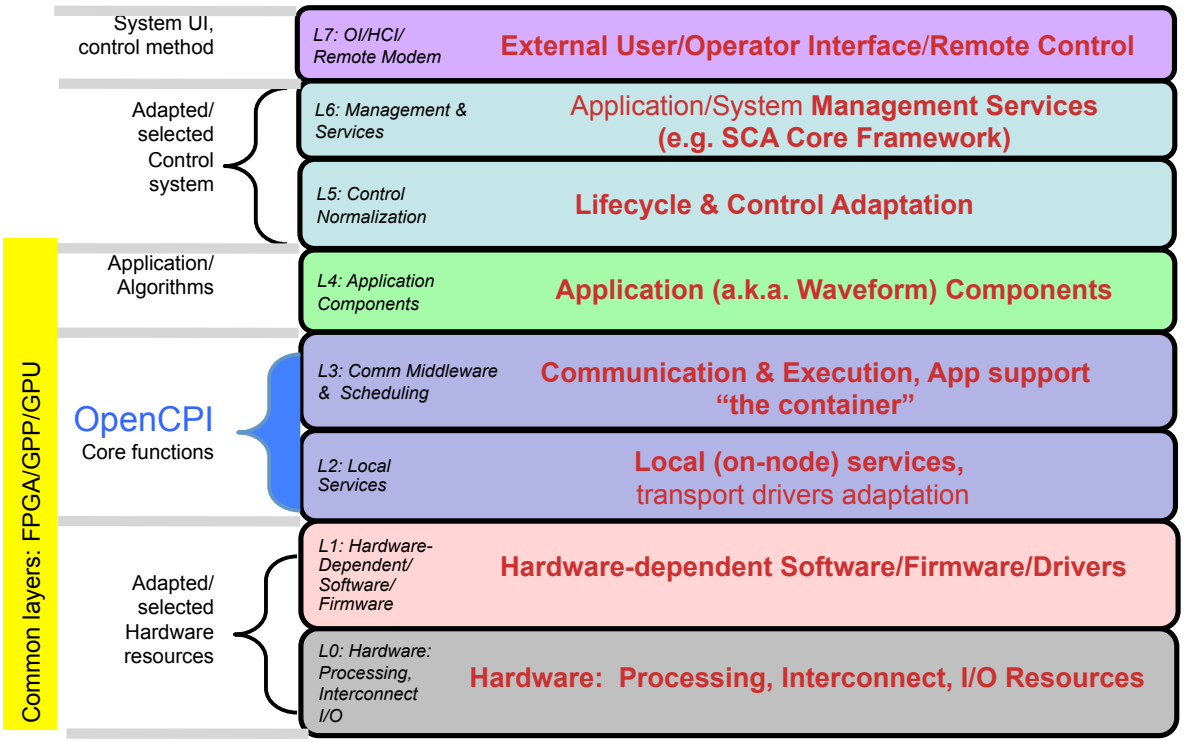

Figure 4: Notional Software Stack

As shown above, level 4 consists of the application components themselves, layers above level 4 are the control and management system. Levels 3 and 2 are where OpenCPI is focused (for container and data transport), and Level 1 is hardware-specific drivers, largely for data transport. OpenCPI supports layers 1 through 3 in software as well as gateware (on FPGAs).

6 Component and Application Development

Developing applications in a component-based environment is cleanly divided into three parts:

- Developing components (using a variety of authoring models)
- Constructing (composing) applications from components (configuring and connecting them).
- Developing a control application that manages the lifecycle and other aspects of the component –based application as a whole.

6.1 Component Development

Most of the effort and machinery provided by OpenCPI for development is for developing components and libraries of heterogeneous components. For each component in an OpenCPI component library, there is an "OpenCPI Component Specification" (OCS) that is an XML-based description of a family of implementations (workers) written in a variety of authoring models. The OCS describes what is in common across all implementations of the same function, which is mostly the external interface features, including the configuration properties (for both initial configuration, and runtime dynamic configuration), and the data ports (and their message protocols). The OCS is also called the "spec file" for the component.

While the OCS for a component is common to all implementations across all authoring models, there is another XML-based description file for *each* implementation (worker), called the OpenCPI Worker Description (OWD), which describes aspects specific to a given implementation, and specific to a specific authoring model.

These two XML files (OCS and OWD) are input to tools that, for each authoring model, can generate various code files (headers, definitions, skeletons), in languages appropriate for the authoring models (C for RCC, VHDL or Verilog for HDL, OpenCL-C for the OCL model etc.). Thus the steps for the start of developing a component are:

- Write the OCS: define the component spec for a group of equivalent implementations.
- Write the OWD: document the key tool-relevant implementation choices specific to the authoring models
- Use tools the generate various source code artifacts for the implementation
- Insert the actual functionality of the component into the source code (called the "business logic" in some CBD texts).
- Build, simulate, characterize, and debug the implementation.

This flow is shown in the following diagram:

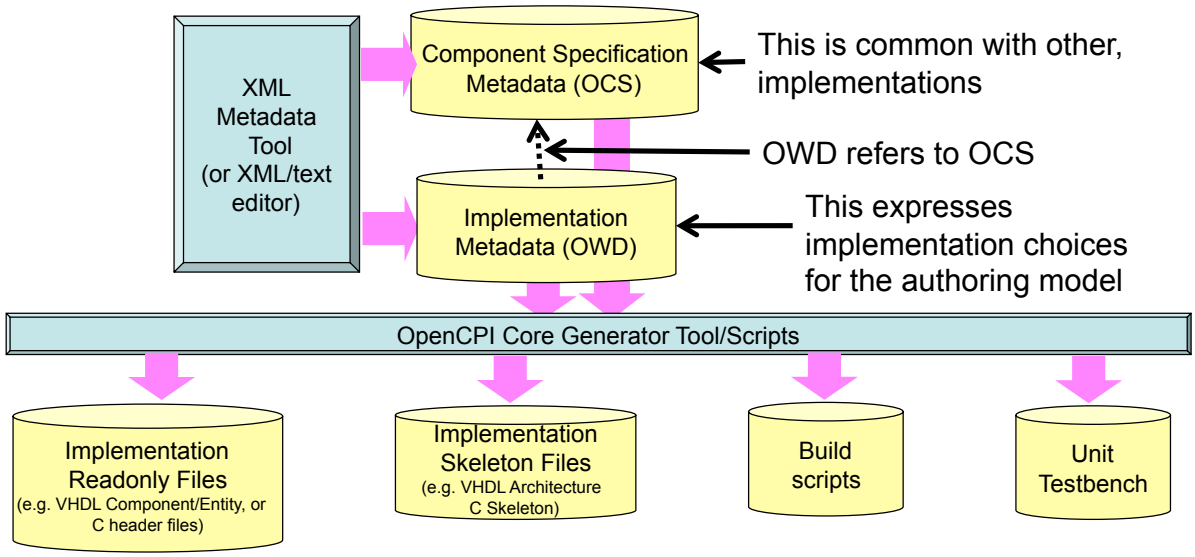

Figure 5: Component Development

6.2 Application Composition

Application composition is simply a composition, or assembly, of components. This involves instantiating a set of components, connecting their ports (or declaring them to be "external" to the application), and providing initial configuration parameters to each component. This composition may be accomplished directly via an "application control interface " or ACI, an API called by a "control application". Or it may be expressed in a simply XML file that does the same three things.

When using an external management framework (such as an SCA "core framework"), there is an adapter code module that enables the framework to launch the application using whatever descriptive means it has.

6.3 The master/control/UI application

While component-based applications can truly be standalone, meaning they just perform their function with no interaction with any other "live software" in any other application or subsystem, it is also common to have other "master" or "control" applications that interact with the component-based application. This is commonly a "control" application that is performing some interfacing function to enable remote or user-interface control of the application. A "master" or "control" application can also decide which component-based applications to run, or run several of them.

In the simplest case there is a simple XML description of the application, and a single generic "launch based on XML" capability.

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 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 scalar values of a 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. They may include hardware resource such registers, memory, and state.

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 worker, while the aforementioned configuration properties are used to specialize components.

Infrastructure – Software/gateware is either application of or 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.