Development of a Component-Based Framework using VSIPL++

Alan Ward, Roger Winstanley, Mark Hayman
Northrop Grumman
{alan.ward, roger.winstanley, mark.hayman}@ngc.com

Brooks Moses
CodeSourcery, Inc.
brooks@codesourcery.com

Introduction

Northrop Grumman is developing a Modular Open Systems Approach (MOSA)-compliant [1] middleware framework upon which to base the software architecture of future net-centric and embedded programs. This presentation will discuss the characteristics of the component-based architecture used in this framework, along with the benefits for design and deployment that such an architecture provides.

We will also describe the importance of the data model to this type of architecture, and in particular the use of the VSIPL++ standard to provide this data model, and some of the extensions added to Sourcery VSIPL++ to support a component-based architecture.

Component-Based Design

The core of this middleware framework is component-based architecture and design. Applications are built from discrete components, which have well-defined interfaces and communicate with each other via both publish-subscribe and request-reply mechanisms. The components may be executed on the same compute node using shared memory, or on different nodes via a network.

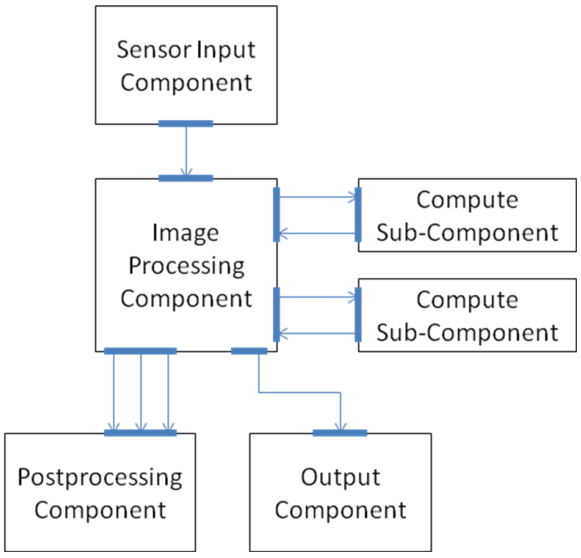

Figure 1: A simplified example of a component layout for an image-acquisition system, illustrating some of the possible configurations. Components may have bi-directional communication with other components, and may receive data and provide data to multiple other components.

The component interfaces are defined using IDL, and these definitions are used to generate C++ code.

Using component-based architecture leads to a number of significant goals for the design and development process:

- **Modularity:** Because component interactions are limited to well-defined interfaces, the components can be independently updated or replaced without affecting the rest of the system.
- **Reuse:** Software is reusable at the component level, rather than only at the full-system level.
- **Interoperability:** Component-based development ensures interoperability between application pieces across multiple platforms.
- **Extensibility:** A component-based architecture is loosely-coupled, supporting easier extension of component and system functionality.
- **Reduced Complexity:** Encapsulation, modularity, separation of concerns and the establishment of hierarchical component dependencies, or "layers", all contribute to reduced design and system complexity.
- **Reduced Design Time:** This reduced complexity and software reusability leads to faster time-to-market and shortened program/software development schedules
- **Lower Design and Maintenance Costs:** The result of shortened design times, modular reusable components, and lowered complexity.
- **Quality and Reliability:** Components can be tested and maintained at the individual component level, instead of only at a monolithic system level.

Portable Deployment

In addition to the development side of the component-based middleware, there is a runtime and execution side based on the DAnCE implementation of the D&C (Deployment and Configuration) specification [2]. Component development and deployment are separated in such a way that components can be developed once and then deployed on different hardware architectures by simply recompiling the source code and modifying the deployment plan.

The VSIPL++ Data Model

In order for components to communicate in a well-defined manner, it is critical to have a well-defined data model. The VSIPL++ standard [3] was chosen as the foundation for the signal- and image-processing within this component-based design because – unlike OpenCL and other signal-processing and math libraries – it is the only multicore signal-processing solution that has such a data model.

In the VSIPL++ data model, data is stored in Block objects, which can have implementation-defined or even user-defined representation in memory, and export a standard set of interfaces. Blocks are wrapped in View objects, which are lightweight objects representing a particular perspective on the data and likewise export a standard set of interfaces. For example, a View may represent an entire data block as a two-dimensional matrix, or it may represent a single row of that data block as a vector.

The VSIPL++ standard also defines a number of signal-processing and mathematical functions that use the data model at a high level, providing an appropriate level of abstraction for applications while allowing sufficient flexibility for an implementation to obtain good performance on a wide variety of hardware.

As a result of this data model, components can be written in a manner which is independent of the underlying data representation in memory, using the standard Block and View interfaces. Meanwhile, the underlying framework and VSIPL++ implementation can choose an appropriate representation for each hardware target; for example, complex arrays may be stored in either split or interleaved fashion depending on the hardware performance characteristics, and systems with coprocessor memory (e.g., GPU-based systems) may use data caching in moving data between memory locations.

Extensions in Sourcery VSIPL++

CodeSourcery has produced some extensions to our Sourcery VSIPL++ library [4] that are necessary for use in the Northrop Grumman component-based architecture.

In particular, although the VSIPL++ standard currently defines an interface for wrapping a Block object around a user-allocated C++ data array, there is not a facility for accessing an arbitrary View as a similar array. This has proved to be necessary for serializing VSIPL++ View and Block objects as IDL objects that can be transmitted across the component interfaces. Thus, Sourcery VSIPL++ has been extended to include methods for this purpose, and CodeSourcery is currently proposing these extensions as additions to the VSIPL++ standard. The flexibility of the VSIPL++ Block model allows for an implementation that can in many cases simply provide a pointer to the internal data, thereby avoiding costly data copies.

In addition, Sourcery VSIPL++ allows users to extend the dispatch mechanism that defines the standard operations on VSIPL++ data. This functionality can be used by the framework programmers to provide additional target-specific implementations of specific functions or combinations of functions, which will then automatically be used by the portable VSIPL++-compliant component programs without any code changes.

Conclusion

A component-based architecture provides a number of significant benefits for development, design, and deployment of large net-centric and embedded signal-processing programs. We have shown how such a system can be built using mainstream market-driven technologies, including in particular the VSIPL++ standard and data model.

References

[1] Open Systems Joint Task Force. “Modular Open Systems Approach (MOSA) Assessment.” [online] http://www.acq.osd.mil/osjtf/mosapart.html.

[2] G. Deng, J. Balasubramanian, W. Otte, D. Schmidt, and A Gokhale. “DAnCE: A QoS-enabled Component Deployment and Configuration Engine.” *Proceedings of the 3rd Working Conference on Component Deployment*, Grenoble, France, November 28-29, 2005.

[3] CodeSourcery, Inc. “VSIPL++ Specification 1.01.” Georgia Tech Research Corporation. 2005 [online] http://www.hpec-si.org.

[4] CodeSourcery, Inc. “Sourcery VSIPL++.” [online] http://www.codesourcery.com/vsiplplusplus.