RASSP Autocoding for Deliverable, real-time Signal Processing Systems (DSP) Control Appnote

Abstract

Deliverable real-time Signal Processing Systems naturally decompose into a Signal Processing Program (SPP) and a Command Program (CP). The SPP is a data flow oriented program that performs high bandwidth numerical processing and typically executes on a suite of Digital Signal Processors (DSPs). The CP is a state-oriented program that executes on a host processor and is responsible for controlling the SPP in response to operator commands. This Application Note will present a simple, layered architecture for the CP and discuss how commercially available autocoding tools and products may be used on all but one of the layers. Particular focus will be placed on autocoding the one layer, known as the Application Specific Interface (ASI) layer, where commercial support is not currently available. It will show how this layer bridges the gap between SPP domain concepts, such as graphs, queues and the data flow paradigm, and the CP domain concepts such as application state or mode. It will also show how this layer exploits the standardized architecture presented by the autocoded SPP. The paper will discuss the software development process in light of the use of autocoding tools for both the SPP and CP. Finally, the paper will discuss the results from using autocoding technology on three RASSP benchmark projects.

Purpose

The purpose of this Application Note is to describe the technical conclusions reached on the RASSP program concerning Command Program autocoding. The Executive Summary and Application Experience sections are appropriate for all readers while the remaining sections are targeted toward a software oriented technical audience.

Roadmap

 Dennis Basara

RASSP Autocoding for DSP Control Application Note

1.0 Executive Summary

Deliverable real-time Signal Processing Systems naturally decompose into a Signal Processing Program (SPP) and a Command Program (CP). The SPP is a data flow oriented program that performs high bandwidth numerical processing and typically executes on a suite of Digital Signal Processors (DSPs). The CP is a state-oriented program that executes on a host processor and is responsible for controlling the SPP in response to operator commands. In large aerospace applications, the CP is frequently viewed as a subsystem of the Command and Control System (CCS). Both programs have traditionally been hand coded and are thus expensive, difficult to integrate, difficult to maintain, and difficult to port to new platforms as the system evolves. But now autocoding a production quality SPP is a reality through the use of tools such as Lockheed Martin's GEDAE™, which provides an advanced environment for designing distributed signal processing applications and lets developers and application programmers graphically construct flow graphs that execute their applications and algorithms. This application note shows how autocoding technology has been pushed into the Command Program domain by exploiting the standardization autocoding tools bring to the Signal Processing Program architecture.

Command Program autocoding technology is in the prototype stage and has been successfully demonstrated on three Rapid Prototyping of Application Specific Signal Processors (RASSP) technology demonstration programs:

1. Benchmark 2 Program, a Synthetic Array Radar application,
2. Benchmark 3 Program, a Sonar application,
3. Benchmark 4 Program, an Image Processing application.

In each application, the technology was used to generate more than 50 percent of the Command Program (CP). The CP development time and the SPP / CP integration time were significantly lower when compared to traditional methods. The autocoded CPs were simple, reliable and were used by the SPP application engineers at the earliest stages of SPP-to-Hardware integration. This technology will be made commercially available as part of the GEDAE™ development environment.

 Dennis Basara

RASSP Autocoding for DSP Control Application Note

2.0 Introduction

This Application Note will present a simple, layered architecture for the Command Program (CP) and discuss how commercially available autocoding tools and products may be used on all but one of the layers. Particular focus will be placed on autocoding the one layer, known as the Application Specific Interface (ASI) layer, where commercial support is not currently available. It will show how this layer bridges the gap between Signal Processing Program (SPP) domain concepts, such as graphs, queues and the data flow paradigm, and the CP domain concepts such as application state or mode. It will also show how this layer exploits the standardized architecture presented by the autocoded SPP. Finally, the paper will discuss the software development process in light of the use of autocoding tools for both the SPP and CP.

The Autocoding for DSP Control application note is organized in sections or chapters as follows:

Section 3.0 is titled "What is a Command Program.#&34 This section provides a definition of the Command Program domain.

Section 4.0, "Top Level Program Architecture", proposes a simple, layered software architecture for command programs.

Section 5.0, "Building the Application Specific Interface", describes how a prototype tool, the Application Specific Interface Builder (AIB), is used to generatethe one layer of the software architecture for which commercial tools are not available.

Section 6.0, "Autocoding Tool Evaluations", reviews tools that could be used for the upper layers of the software architecture.

Section 7.0, "Software Develop Process", presents a software development process for building command programs.

Section 8.0, "Application Experience", describes the results of utilizing the ideas, tools and processes presented in the previous sections on three RASSP benchmark projects.

 Dennis Basara

RASSP Autocoding for DSP Control Application Note

3.0 What is a Command Program

A typical large-scale embedded signal processing architecture consists of a signal processing system communicating with a command and control system. The signal processing system performs the high-bandwidth numerical processing and typically executes on Digital Signal Processors (DSPs). It is controlled by a command program (CP) that is intimately related to the signal processing system. The command program serves as an interface between the signal processing program and the rest of the command and control system. This is done by translating system-derived or user provided inputs into commands understood by the signal processing system. The results processed by the signal processing system are then fed back to the command and control system. The command program serves as an interface between the signal processing program and the rest of the command and control system. This is done by translating system-derived or user provided inputs into commands understood by the signal processing system. The results processed by the signal processing system are then fed back to the command and control system. Figure 3 - 1, A Typical Large Scale Aerospace Architecture, shows the command program ambiguously residing in both the Signal Processing System and the Command and Control System. The generic controls over the signal processing program all reflect the graphical structure of that program, and thus the CP is frequently conceptualized as a part of the Signal Processing System. But this is only half of the story. The CP receives commands and forwards results to the Command and Control System, which is unaware of the graph structure of the Signal Processing System. This communication between the CP and the rest of the Command and Control System (CCS) occurs in the "language" of the CCS which usually is state oriented. For example the CCS may want to place an Airborne Radar Signal

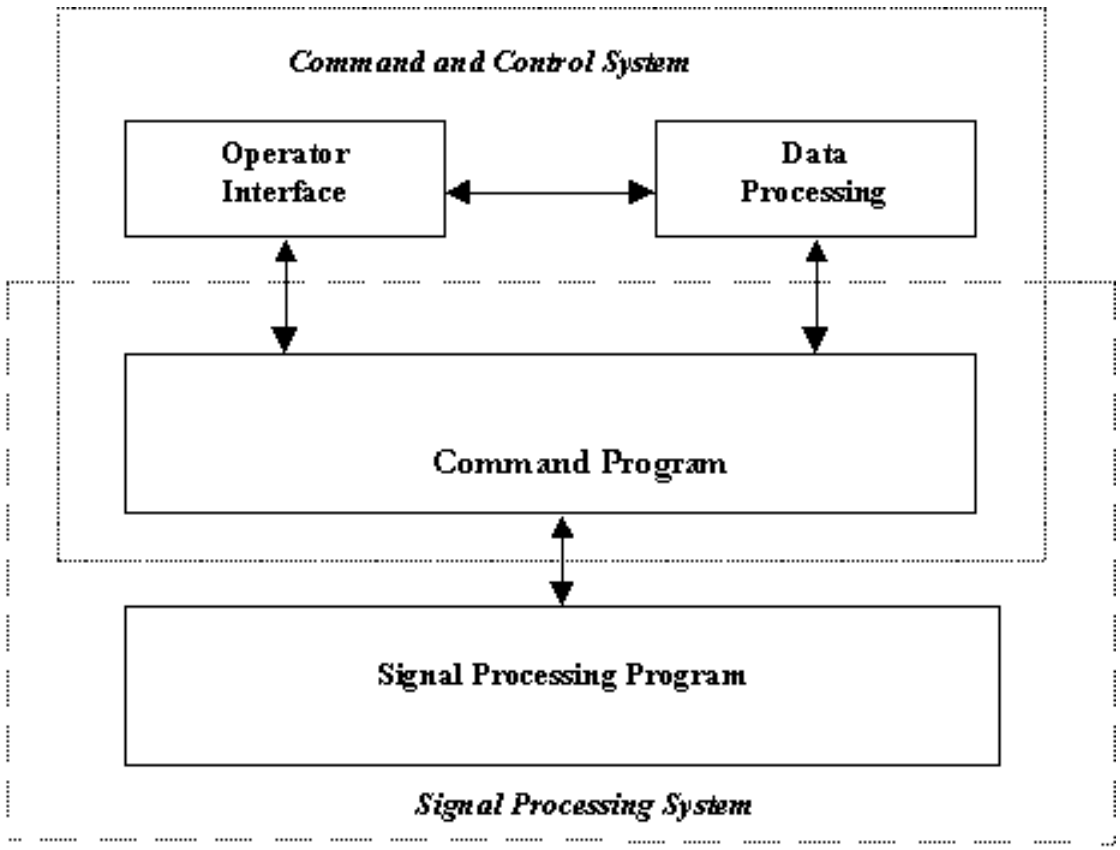

Figure 3 - 1: A Typical Large Scale Aerospace Architecutre

Processing System into the Air-to-Ground mapping state. Since the CP also speaks this state oriented language it may be viewed as part of the CCS. Control a Signal Processing Program that has been graphically specified via a dataflow model of computation and autocoded, a generic set of controls over the signal processing graphs are made available to the CP. They define the complete set of commands understood by the signal processing program. (See the Data Flow Graph application note for a complete discussion of these capabilities.) The abilities provided fall into several broad categories:

- Graph instantiation - comprises loading signal processing program executables onto their associated suite of processors as well as obtaining handles for parameter and queue objects.
- Runtime program to processor mapping - maps SPP executable programs to DSP processor.
- Execution Control - commands that allow the CP to control the execution state of the signal processing programs. The SPP execute asynchronously to the CP.
- Data flow control (limited) - includes control of the standard process states such as run/stop/reset, as well as the ability to control the flow of input data into the signal processing graph. Because of the data-flow paradigm of the graph, this determines whether a "running" graph is stalled or executing,
- Write access to graph parameters - The CP is responsible for writing graph parameter data to the signal processor. Since graph execution is asynchronous to the CP, the CP must carefully control the graph execution state before writing data via graph parameters.
- Read/Write access to graph queues connected to the CP - allows CP to asynchronously read data from graph or write data to a graph.

Depending on real-time conditions, some data-flow may need to be rerouted either within a graph or among distinct graphs and is therefore also under the control of the CP. It is important to emphasize that most data-flow is not under control of the CP. That which is under CP control is used to perform infrequent, significant graph reconfigurations. The CP may manage data-flow within a graph by simply setting parameters controlling "switches" in the graph. Data flowing between distinct graph instances may be managed by connecting and disconnecting dynamic queues serving as pipes between the graphs. The CP can also be connected via dynamic queues to the signal processing program. Typically the data read/written on these queues is low rate.

RASSP Autocoding for DSP Control Application Note

4.0 Top Level Command Program Architecture

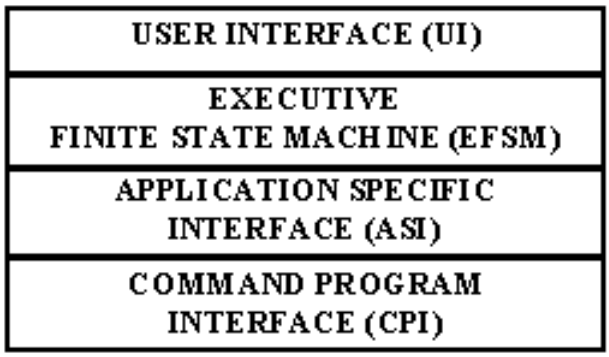

Figure 4 - 1: Layered Command Program Architecture

Figure 4 - 1, Layered Command Program Architecture, reflects the bridging role between the CCS and the SPS played by the command program. The bottom two layers address the Signal Processing Program perspective while the top two layers address the CCS perspective. The bottom most layer is the Command Program Interface (CPI) layer and consists of the generic set of graph control functions provided by the Signal Processing Application autocoding tool. This layer can be viewed as Commercial of the Shelf (COTS) software from the perspective of the CP. The next layer is the Application Specific Interface (ASI) layer. This layer provides a set of application specific low level control functions. For example, where the CPI level might provide a generic "appParamWrite" function, the ASI might provide an application specific "writeMySpecificParam" function for each CP visible parameter in the application. This layer is discussed at length in Section 5.0, Building the Application Specific Interface.

Command programs are highly state oriented programs since they control and reflect the state of the signal processing application. For example, an airborne radar application CP may include an airborne target search mode and an airborne target track mode. The Executive Finite State Machine (EFSM) layer captures the CP state functionality and manages transitions between states. As discussed in Section 6.0, Autocoding Tool Evaluations, commercially available autocoding tools may be used to construct this layer of the CP. The final layer is the User Interface layer. It manages communication with the remainder of the Command and Control System. For some applications this layer may consist of the Graphical User Interface (GUI) code. The RASSP command program autocoding project did not address this layer. The layered nature of this architecture is illustrated in 4 - 2 through Figure 4 - 4.

A User Interface (UI) layer function that reads a state command from the user is illustrated in Figure 4 - 2, User Interface Layer Calls Executive State Machine Layer. The command is passed to an Update State routine that resides in the EFSM layer.

```
UI.getUerInput() {
    StateCmd = ReadStdIn() ;
    ESM.updateState(state_cmd) ;
```

Figure 4 - 2: User Interface Layer Calls Executive State Machine Layer

This EFSM layer routine, shown in Figure 4 - 3, Executive State Machine Layer Calls Application Specific Interface Layer, validates the legality of the state transition requested and then invokes a sequence of ASI layer functions to accomplish the request.

```
ESM.updataState(state_cmd) {
switch ( mode_state ) {
    case: modeA
    switch ( submode_state ) {
       case: submode1
       switch ( state_cmd ) {
           case: go_to_modeB
            ASI.Pause () ;
            ASI.reParam1 (outdata ) ;
            ASI.KillModeA() ;
            ASI.EnterModeB() ;
```

Figure 4 - 3: Executive State Machine Layer Calls Application Specific Interface Layer

The ASI layer routine invoked is shown in Figure 4 - 4, Application Specific Interface Layer Calls Command Program Interface Layer, and is called EnterModeB. This is the routine that directly manages the graph. It invokes a collection of CPI layer calls to obtain handles for the new graph, initialize the hardware, load the graph etc.

```
ASI.EnterModeB() {
    apModeBHdl = AII.appRead("ModeB.Graph.Name")
    CPI.initLaunchPackage (apModeBHdl) ;
    CPI.initEmbSolaris() ;
    CPI.appLoad(apModeBHdl) ;
    CPI.appReset(apModeBHdl) ;
    ApParam1Hdl = CPI.appGetParam (apModeBHdl, "Graph.Name.of.Param1") ;
```

Figure 4 - 4: Application Specific Interface Layer Calls Application Independent Interface Layer

The complexity of the various CP layers will vary with the application. For some simple CPs the Electronic Finite State Machine layer might be absorbed into the Application Specific Interface (ASI) layer and/or the User Interface (UI) layer. In a multi-mode radar application, the ESM layer will be extensive since the Command and Control System will need to sequence the Signal Processing System through multiple, possibly concurrent or nested states. If a collection of graphs presents a complex interface to the CP with many graph parameters and dynamic queues, the ASI interface layer will be extensive.

Next Up Previous Contents

Next: 5 Building the Application Specific Interface **Up:** Appnotes Index **Previous:**3 What is a Command Program

 Dennis Basara

RASSP Autocoding for DSP Control Application Note

6.0 Autocoding Tool Evaluations

Under the RASSP program several commercially available tools for producing the Executive State Machine layer of the CP were evaluated. Key evaluation criteria included the ability to easily and clearly graphically model finite state machines based applications, perform functional simulation, import externally generated library routines, and generate efficient, production quality source code. The tools evaluated were SystemBld by Integrated Systems Inc (ISI, maker of MatrixX), BEACON by Applied Dynamics International (ADI) and ObjectGEODE (OG) by Verilog. Verilog's ObjectGEODE was selected as the most suitable, commercially available tool, as shown in Table 6 - 1, EFSM Level Tool Evaluations.

CRITERIA	ObjectGEODE	SystemBld	Beacon
FSM Graphical Representation	Good	Good	Poor
Overall Graphical Representation	Good	OK	Good
Functional Simulation	Yes	Yes	No
Importing Source Code into Generated Source Code	Easy	Hard	Easy
Code Generation	Good	Poor	Good

Table 6 - 1: EFSM Level Tool Evaluations

- The SystemBld tool was the least suitable for use in the Command Program domain, as it is traditionally a tool targeted for *classical* control systems. As a consequence the graphical representation of the overall command program was very awkward. Additionally, it was difficult to import externally generated functions.

- Even though Beacon was not selected as the recommended tool, it had several important strengths with regard to command program generation. Its code generation in both Ada and C was terse and clear. Its graphical interface was similar to classical flow charts and consequently very closely represented the generated code. The down side to the low level graphs is that the abstraction level that the engineer works at is not raised, thus reducing the benefits of autocode generation. The up side to the low level graphs is the control over the generated code provided to the engineer.

- ObjectGEODE is dedicated to analysis, design, verification, and validation through simulation, code generation, and testing of real-time and distributed applications that are best represented by a finite state machine model of computation. It targets real-time applications based on the Specification Design Language (SDL), a European International Telecommunications Union Technical committee (ITU-T) Standards Body, finite state machine modeling language that is textually and graphically defined. ObjectGEODE provides graphical editors; a powerful, interactive, random and exhaustive simulator; a C code generator targeting popular real-time OS and network protocols; and a design-level debugger.

As part of the RASSP program, tool enhancements to ObjectGEODE that were commercially-viable and desirable for command program generation, were identified and funded. These enhancements are as follows:

- The OG execution trace capability was extended to allow easy tracing of the ASI level function calls.
- The OG graphical support for hierarchical states was extended, since Modes and Submodes are hierarchical. The approach to hierarchical state modeling implemented by Verilog is being proposed to the ITU-T for incorporation into the SDL standard.

- Verilog developed a method for incremental migration of OG models to target hardware.

Most of these enhancements will be commercially available in version 1.2 of ObjectGEODE.

The use of ObjectGEODE will prove most beneficial on large command programs with complex state transition logic. It manages program distribution effectively by allowing the designer to explicitly identify and map program processes. OG uses an extended finite state machine model which allows for complex state transitions to be modeled using a flowchart like syntax.

RASSP Autocoding for DSP Control Application Note

7.0 The Software Development Process

To meet demanding development schedules, concurrent command program and signal processing application development is critical. The hierarchical nature of the graphically defined signal processing programs and the use of AIB facilitate this critical parallelism. Figure 7 - 1, Software Development Process, shows the development process model. It has as its root the top-level data flow graph definition. It is natural and good engineering practice for the DSP application designer to produce this graph early in the design process. In most cases, the basic structure of well designed top level graph or graphs will remain fairly stable throughout the project. Most variation will occur in the number or data type of interconnecting queues and control parameters. Since the ASI layer contains these details, the use of AIB mitigates the impact of these variations.

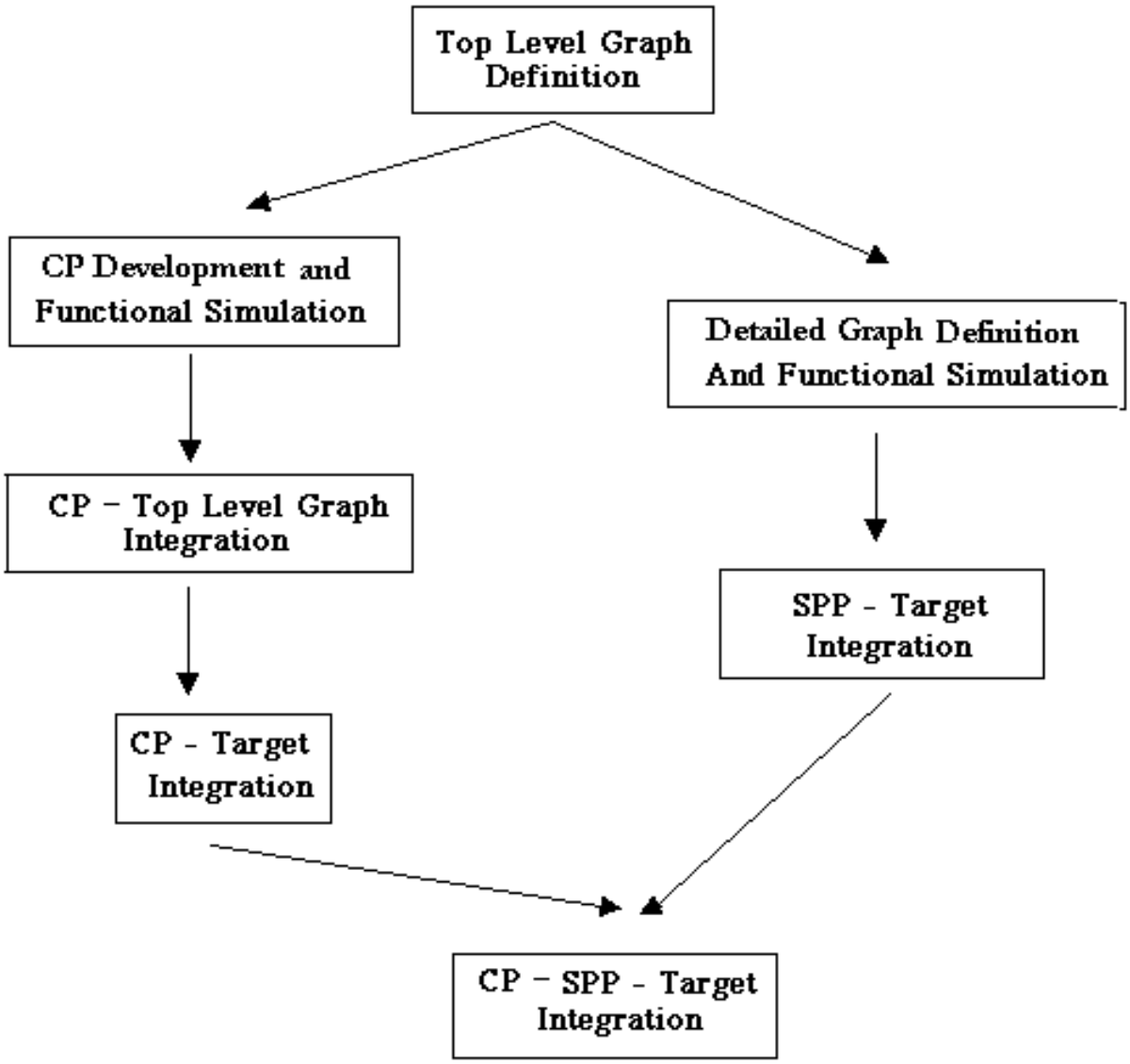

Figure 7 - 1: Software Development Process

STEP 1: Command Program Development and Functional Simulation

Command Program development starts with overlaying the application mode/submode structure onto the top level graph definition, which enables the execution of AIB. The mode/submode structure of an application will frequently be clearly contained in the application top-level specifications, so after some iteration with the DSP application designer the mode/submode structure can be defined quickly. In fact, it is anticipated that when AIB is incorporated into the DSP application development environment, it will be the responsibility of the DSP application engineer, not the CP engineer, to define the mode/submode structure and to invoke AIB.

Once the Application Specific Interface has been generated by AIB, the construction of the Enhanced Finite State Machine (EFSM) and the User Interface (UI) layers may begin, since either might invoke AIB generated functions. If the EFSM layer is complex, ObjectGEODE (OG) might be used at this stage and if a GUI is required, a GUI builder might also be used to generate the UI. Following construction of the OG model of the CP, it can be exercised using the functional simulator provided by the tool. Since the EFSM layer embodies the heart of the CP application state functionality, this step allows the functionality of the CP to be validated prior to code generation.

STEP 2: Command Program Top Level Graph Integration

Following CP development and functional simulation, the CP is integrated with the top level graph. The Top Level graph at this stage performs only data flow adequate to exercise the command program. Since graphs can easily be retargeted in GEDAE™ this integration can be performed on the CP development host.

STEP 3: Command Program Target Integration

The goal, at this development step, is to verify that the actual command program from the ASI layer up executes as expected with the target hardware and in cooperation with the rest of the command and control system. This is performed independently from the detailed signal processing. The approach is to again use the Top Level graph but to retarget its execution to the final target hardware; this is easy to do in the GEDAE™ development environment. The new launch package will need to be run though AIB, although the ASI layer interface will remain unchanged insuring that higher levels remain untouched.

STEP 4: System Integration

Concurrent with Step 3, Command Program Target Integration, DSP application target integration has been occurring. When this critical step is complete, the final system integration uniting CP and SPP may occur. The experience with the RASSP Benchmark projects showed that this integration step was significantly simplified as a result of the standardization inherent to autocoding technology.

Next Up Previous Contents
Next: 8 Application Experience **Up:** Appnotes Index **Previous:**6 Autocoding Tool Evaluations

RASSP Autocoding for DSP Control Application Note

8.0 Application Experience

The Application Specific Interface Builder (AIB) was used on both Benchmark (BM) 3 (Sonar application) and Benchmark 4 (Image Processing application). A very preliminary version was used on a simplified Benchmark 2 (Radar application). In all cases AIB was used to generate more than 50% of the CP source code. The mode / submode abstractions proved appropriate for each benchmark domain and Application Specific Interface (ASI) level software did not need to be tailored for any of the examples. None of the Benchmark CPs were distributed applications nor did they involve complex mode/submode level state transition logic. Consequently, ObjectGEODE was not employed to generate the Enhanced Finite State Machine (EFSM) layer of any of the CPs.

8.1 BenchMark 2

As an initial proof of concept, a simple version of AIB together with ADI's Beacon tool was used to generate a simplified version of the BM 2 CP. The Signal Processing Program (SPP) was generated by MCCI's signal processing autocoding tool. At the time of the experiment, the MCCI Command Program Interface (CPI) was not completed so a simple CPI allowing the CP to run / stop / load a parameter was created by modifying a similar existing interface. Since the CPI and UI level software were in Ada, the proof of concept AIB was programmed to generate Ada. The Beacon tool, which also had an Ada generation capability, was used to generate the EFSM level. The combined autocode CP and SPP application was successfully demonstrated at the November 1996 RASSP conference.

Layer	Generation Tool	Lines (approx)
User Interface (UI)	Heritage	200
Execute Finite State Machine (EFSM)	Beacon	100
Application Specific Interface (ASI)	AIB	500
Command Program Interface (CPI)	hand	?

Table 8 - 1: Benchmark 2 Source Code

Following the success of the BM 2 proof of concept AIB was rewritten to generate C compatible with the GEDAE™ CPI. This version was used on both BM3 and BM4.

8.2 BenchMark 3

The Benchmark 3 (BM3) program provided the opportunity to demonstrate AIB on a Mode with multiple submodes and to extend the code generated by AIB into the UI and EFSM layers. As can be seen from Figure 8 - 1: Schematic BM3 Top Level Graph, the BM3 SPP appears to naturally split into two submodes depending on the Switch Box control parameter. During graph testing however it became clear that BM3 should in fact be viewed as having four submodes. This is due to four distinct parameter sets being developed and the graph pipeline needing to be completely cleared of processing before changing each parameter set. In essence the CP path had two sub-configurations and the CW path also had two

sub-configurations. AIB generated ASI layer software easily accommodated this change in perspective.

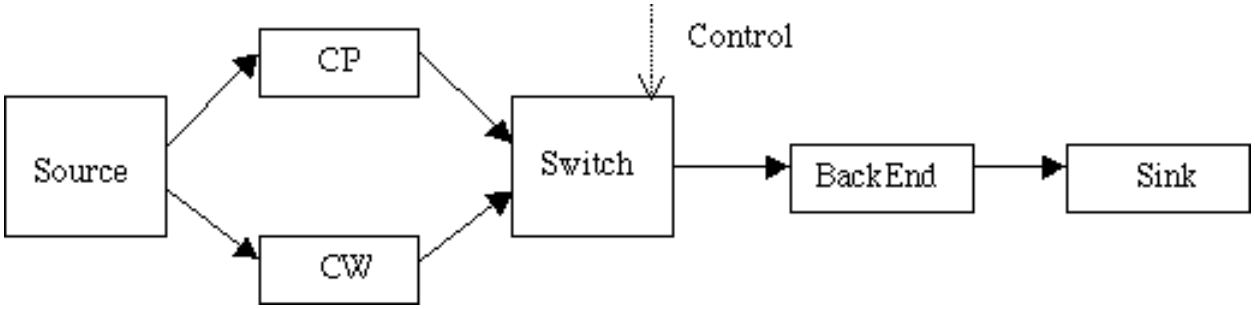

Figure 8 - 1: Schematic Benchmark 3 Top Level Graph

Although the BM3 application CP is ultimately intended to be embedded in a large UYS-2 CCS, for the acceptance test the CP needed to be a simple stand-alone program. This provided the opportunity to extend the scope of code generation in AIB to the UI and EFSM layers. AIB was extended to generate a "generic" CP where the User Interface was textual and was composed of a collection of menus that allowed the selection of submode and setting of all parameters visible to the CP. This generic CP proved to be sufficient, with minor tailoring, for the BM3 acceptance test. As can be seen from Table 8 - 2, BM3 Source Code, the combination of AIB generated ASI software and the AIB generated generic CP reduced the amount of handwritten software for the CP to less than 10%.

Layer	Generation Tool	Lines (approx)
User Interface (UI)	AIB	1200
Execute Finite State Machine (EFSM)	hand AIB	200 200
Application Specific Interface (ASI)	AIB	3500
Command Program Interface (CPI)	GEDAE COTS	750

Table 8 - 2 Benchmark 3 Source Code

Additionally, a second simplified BM3 command program was constructed using ObjectGEODE. The purpose of this experiment was to validate the ability to easily integrate AIB generated source code, ObjectGEODE generated source code and the GEDAE™ CPI into a single command program to control a GEDAE™ graph. The results showed that this process was straight forward.

8.3 BenchMark 4

Benchmark 4 (BM4) provided the opportunity to verify that AIB generated software could cross application domains and be easily integrated into a command program largely composed of heritage software. The BM4 program sponsor, MIT Lincoln Laboratory, provided an executable specification and test bench for the application. The test bench included a process known as "highClass" that controlled a process known as "Candidate". Most of the processing in Candidate was to be moved to 72 SHARC processors with the SPP generated by GEDAE™.

Layer	Generation Tool	Lines (approx)
User Interface (UI)	Heritage	400
Execute Finite State Machine (EFSM)	Hand	100
Application Specific Interface (ASI)	AIB	500
Command Program Interface (CPI)	GEDAE COTS	750

Table 8 - 3: Benchmark 4 Source Code

Since highClass already possessed all of the UI and EFSM level code to interact with the rest of the test bench, it was natural to transform highClass into the command program. This transformation process proved straight forward and largely consisted of removing software that had interfaced with Candidate, moving some Candidate software into highClass, restructuring the main program loop and adding calls to ASI level software. The code sizes shown in Table 8 - 3, Benchmark 4 Source Code, indicate that approximately 50% of the command program was constructed from AIB generated software. Note that AIB generated additional functions, totally another 1000 lines, were not needed for this particular application. Also note that a significant amount of Interprocessor Communication Software employed to communicate between the CP and the rest of the software test bench is not included in Table 8 - 3. (It was reused without modification.)

A number of conclusion can be reached from the RASSP benchmark projects:

- The Application Specific Interface Builder (AIB) simplifies CP construction by autocoding the Application Specific Interface layer that is error prone and tedious to implement manually. The ASI layer is needed by all command programs.
- The Application Specific Interface Builder generates an easily tailored generic CP that can serve as a basis for demonstration and test command programs.
- ObjectGEODE generated code is easily integrated with AIB generated code to form CPs that control GEDAE™ generated graphs.

Additional work is required on the Application Specific Interface Builder to prepare it for inclusion with the GEDAE™ commercial product line. The ASI layer abstractions require further refinement and the generic command program concepts need to be extended. Some of this work will be performed in 1998 and 1999 under the Navy/Lockheed Martin ATL DUAP program.

Next Up Previous Contents
Next: 9 References **Up:** Appnotes Index **Previous:**7 The Software Development Process

RASSP Autocoding for DSP Control Application Note

9.0 References

Lee, E.A., "Integrating Control and Signal Processing, Report for Lockheed Martin, Advanced Technology Laboratories (ATL)", University of California, Berkeley, July 6, 1997. http://ptolemy.eecs.berkeley.edu/~eal/reports/atl.final.pdf

Fickle, C., "Evaluation: Beacon Product Family for Signal Processing Command Program Generation", Lockheed Martin Advanced Technology Laboratories, August 1996. [BEACON_96]

Fickle, C., "Evaluation: MATRIXX Product Family for Signal Processing Command Program Generation", Lockheed Martin Advanced Technology Laboratories, August 1996. [MATRIX_96]

Verilog (ObjectGEODE)
http://www.verilogusa.com

Applied Dynamics International (BEACON)
http://www.adi.com

Integrated Systems Inc. (systemBld, MatrixX)
http://www.isi.com

Application Notes

Data Flow Graph Design

 Dennis Basara