EVALUATION: MATRIX$_X$ Product Family for Signal Processing Command Program Generation

by C. Fickle

Evaluation Summary:

The MATRIX$_X$ Product Family is a mature tool set that can successfully be used to graphically model and autocode generate Ada source code for signal processing command programs whose primary function is to control graph execution and configuration variables. The code generated is not comparable to hand written code in terms of clarity or efficiency, but is adequate for real-time applications that are not severely CPU or Memory constrained. Conversion between the limited set of model data types and the extensive set of Ada data types provides a source of potential errors during both program creation and program maintenance. The product has the potential to be tailored into a stand alone command program simulation environment and the potential for integration with signal processing graph tools to facilitate the rapid development of command programs.

Command Program Requirements:

The command program used for this evaluation was required to control, at the behest of an operator, three communicating signal processing applications that were representable as PGM graphs. The signal processing application provided a set of graph support services, encapsulated within a single Ada package, that included starting and stopping a graph, creating and controlling external graph IO processes, creating queues used for inter-graph communication, and creating, setting, and reading graph variables used to configure graph execution. There was also set of target hardware control services similarly encapsulated. The Command Program served as mediator between the graph support services and hardware support services, and a GUI that provided the operator control over the entire application, including the capability to RUN, STOP, REBOOT, LOAD and DUMP graph variables, and TEST the hardware.

A command program previously hand written in Ada served as detailed functional requirements. No effort was made to replicate the structure of this program.

The target processor for the control program was a 68040 running Vxworks. The development platform was a SUN workstation running SunOS 4.1.3.

Tools Used:

The MATRIX$_X$ Product Family, provided by Integrated Systems, Inc. (ISI), is a suite of tools developed to analyze, graphically define, simulate, and autocode generate classical control system applications. The tool was chosen for this application because it supported several key features: finite state machine editing, Ada-83 autocoding, external Ada package importing, autocode generator tailoring. The tools used for Command Program generation follow:

1. **SystemBuild**: A graphical editor used to represent an application was used extensively to develop the command program model. The tool is designed for use for the simulation and development of *classical* control applications, hence most of the Blocks provided on the Block palette were not used.
2. **Autocode**: A translation tool used to transfer a SystemBuild model representation into compilable code was used extensively to generate stand-alone Ada-83 code. The tool also provides C-code generation, which was not utilized.
3. **Xmath**: An analysis tool optimized to perform matrix calculations was used to start SystemBuild and to review Autocode output messages.

Building the Application:

The following steps were taken to build the evaluation application.

1. ***Define a User Code Block (UCB) for each visible procedure used from the Graph Support Package and the H/W Support Package.***
2. ***Encapsulate each UCB within a Procedure Superblock to allow the UCB to be referenced in conditional control blocks.***
3. ***Build the Command Program model using the SystemBuild tool.*** There were several key parts to the Command Program model. A collection of data stores was defined such that each data store encapsulated the attributes of a graph object. For example, the data store for each graph variable defined a numeric identifier, a data type, a scalar/array indicator, initial value or array selector, and storage for a value loaded via the user interface. A finite state machine was built that issued procedure activation triggers on state transition boundaries. Finally a collection of procedures were constructed that contained conditional control blocks that would sequence encapsulated UCBs.
4. ***Build an Ada wrapper around each of the support packages that translates the autocode generated interface for each of the UCBs to the actual interface expected by the support package software.*** At real-time autocode generator runtime, the tool has no knowledge of the specifications for imported procedures. It knows only procedure names. As a result the generator assumes a standard specification for each imported procedure.
5. ***Build Ada package specifications to enable translation between model data types and the services packages, since the data typing supported by the tool and autocode generator is spartan.*** Two Ada packages were created; One provided a set of object name strings, and the other, defined a collection of arrays that could be referenced by number rather than name or pointer.
6. ***Build an Ada wrapper around the existing user interface software to translate model external inputs and outputs.*** This included writing a simple Interface Control Document defining this translation.
7. ***Build a simple Ada Message package that allows the Command Program model to output messages to standard output.*** No wrapper was required for this package since its interface was designed to be compatible with the autocoded UCB.
8. ***Modify the autocoding template provided with the tool suite.*** The wrapper and message packages needed to be “withed” in several location and the default external interface procedural references needed to be replaced by calls to the new User Interface wrapper. An interface task was defined to decouple the external interface from the rest of the program.

9. ***Autocode, compile, link, execute and debug program.*** The graph support services package body was replaced by debug code for stand-alone testing. The compiled code was readable, if not elegant or terse. Debugging at the source code level while implementing changes at the model level was at times frustrating. Part of the learning curve was understanding how the code generator handle various constructs.
10. ***Integrate with graph application and target processor.*** The initial version of the model was structured such that the code generator made extensive use of Ada tasking. However, during hardware integration it was discovered that the communication services between the command program and the graph manager required that all communication emanate from a single task. This resulted in a significant restructuring of the model.

Statistics:

The resulting program consisted of approximately 5500 lines (egrep -e ';' *.a | wc -l) containing a semicolon (;). This is about a 50% increase over the hand written version used as a detailed functional requirements baseline. The handwritten wrapper and support programs accounted for approximately 1000 of the 5500 lines. Approximately six man-weeks were required to perform the development task from inception through hardware integration. An additional one week was spent on tool training provided by the tool vendor.

Most Subtle Problem:

Subsystems 1 and 2 both write to data store A. Subsystem 1 is of higher priority than subsystem 2 and sets A = 1. Subsystem 2 executes and sets A = 0. Subsystem 1 executes again and A has a value of 1!. The apparent reason: The data store was not really written to until the subsystem outputs were posted. In the loop that posted outputs, subsystem 1 outputs were posted after subsystem 2 outputs.

Enhancements to the Tool Suite:

The tasks noted below would enhance the ability of the $MATRIX_X$ family of tools to support creation of Command Programs. These tasks require modification of the tools and would therefore need to be performed by the vendor.

1. ***Expand the set of data types that can be passed to a User Code Block.*** The most significant weakness of the tool for this application was the limited set of data types provided. The interface between code generated from the model and a wrapper consisted, primarily, of a real input array and a real output array. Thus the wrapper needed to provide the correspondence to strings, array names, and enumerated types used in the Ada service packages, thus negating the benefits of traditional symbolic naming.
2. ***Allow arrays as external inputs and outputs.*** Currently aggregate types are not allowed as external inputs or outpus.
3. ***Allow blocks to be added to the tool palette.*** The set of encapsulated UCBs for the graph support and hardware support packages define a set of primitive instructions for Command Program programming, not unlike the set of primitives defined by the tool for classical control applications. This would allow the tools to be tailored to a specific class of applications.
4. ***Allow new classes of block interfaces to be recognized by the autocoding tools.*** This would

allow bypassing the standard UCB interface and may obviate the need to construct wrapper programs if a collection of procedures have a consistent interface.

5. ***Provide execution time and code size estimates for autocode for selected targets.*** This is important information for early software and hardware design tradeoffs.
6. ***Support large scale developments.*** Currently the autocode output is written to a single file. Large projects will typically need the ability to configure and track source code changes at the package level. Most large projects would have a different person working on each subsystem, so there would be a need to be able tie together multiple models. The portion of a model developed by each individually would need to be individually autocoded for stand-alone testing.
7. ***Provide a visual means to specify relative execution order within a triggered superblock for a collection of procedures that do not have data flowing among them*** Frequently a collection of independent procedures needed to be executed within a single triggered superblock in a specific order. Since there was no data flowing between the various procedures, the only means to guarantee execution order was to pass both the trigger and the input / output data for all the procedures through a conditional control block. This resulted in complex diagrams.
8. ***Provide A Multiple Procedure Condition Block.*** Frequently multiple procedures needed to be conditionally executed based on the same condition signal. The condition block provided with the tool required the condition signal to be input for each "if - endif" pair. This resulted in unnecessarily complicated figures.
9. ***Visually clarify relationship between condition block I/O and referenced procedures.*** Currently a Condition block can be displayed showing the set of procedures referenced by the block. However the connection menu needs to be pulled up to show the correlation between condition block inputs and procedure inputs. The diagram information content would be significantly expanded if a line was shown from the an input to its respective function.
10. ***Eliminate typing of procedure names and I/O counts in Condition Block Popup Window.*** Currently precise recollection of procedure block name and I/O is required. For example a pull-down menu with all known procedure names would be helpful.
11. ***Preserve condition block association between input, function, and output.*** When a new function is added, or the I/O is changed on an existing function, to a sequential condition block the association between inputs, outputs, and functions frequently was no longer valid.
12. ***Provide a case structure primitive.*** To fetch an ID from one of five data stores as a function of a selection number provided from the external interface, a true/false signal had to be generated for each possible selection value and then fed into a cascading set of data flow switches.
13. ***Allow datastores in procedural blocks.*** Even though data stores are treated as global objects they should be allowed to be graphically placed within a procedural block, even though the generated code has the datastore value being passed into the procedure. This will allow blocks with significant references to data stores to be declared procedural blocks and thus referenced within condition blocks.
14. ***Allow multiple writes to the same datastore from within a single subsystem.*** This is currently flagged as an error. But a subsystem should be allowed to have disjoint execution paths, with each path setting the same data store.
15. ***Provide control of which triggered blocks become tasks.*** Currently if a block is triggered by a specific signal (opposed to by "parent") the block will be come a task, which proliferates Ada tasks. Allow a triggered block, at the discretion of the model designer, to be simply a conditionally called block.

16. ***Provide model designer visibility into the resulting set of subsystems.*** Currently to determine which blocks are placed each subsystem, requires the designer to carefully review each block type and frequency. The assignments are not even centrally commented in the resulting code. Subsystem assignment is important for the designer to be aware of because the subsystem become program independent program threads.
17. Eliminate requirement for top level block to have an external output.
18. ***Provide a mechanism for global changes.*** Programmers frequently make (careful) global changes to programs they have written. No comparable mechanism is provided. For example, to change all discrete blocks with a frequency of 2 seconds to a frequency of 1 seconds requires pulling up the associated menu for each block.

Integrating with RASSP:

The tasks noted below would tailor the MATRIX$_X$ tool environment for use for command program generation. These tasks can be performed without modification to the MATRIX$_X$ family of tools.

1. ***Provide a stand-alone command program simulation environment within the MATRIX$_X$ tool suite.*** The tool suite provides considerable simulation capability including input generation and output plots. A SystemBuild model references UCBs only by file name and function name, allowing the same SystemBuild model to be used as the source for generation of Ada real-time code and as the source for the simulation environment built using C code. Since the file name is ignored during real-time *Ada* autocode generation, the file name can point to a C program that provides debug code for each of the graph support and hardware support routines. (Note, that this debug code must be written in C, since only C routines can be imported into the MATRIX$_X$ *simulation* environment.) The simulation is entirely self contained within the tools since even messages to be output by the model can be directed to the MATRIX$_X$ window, which supports session logging. It should be noted however that validating a model within the simulation environment does not validate the Ada code generated by the real-time code generation tools, since the other major input to the autocoding process, besides the model, is the autocoding template which of course varies with language.
2. ***Integrate with signal processing graph generation tool.*** A considerable portion of the work required to generate the command program used to evaluate this tool involved defining data stores for the graph objects and basic command scripts for graph start and stop, graph variable creation and initialization, and queue creation. Much of this work could be automated by using a scripting language the MATRIX$_X$ tool family provides that can be used to generate SystemBuild models. Tools could parse the output of the graph generation tool (e.g. the SPGN output by a PGM tool) to identify the graph objects, such as top level graphs and graph variables, which could be input into a script generation tool. The resulting MATRIX$_X$ script would be executed by Xmath to generate a basic command program model. The model may provide rudimentary command program functionality for initial testing of graph sequences, but more importantly, would serve as the prototype from which to evolve the final command program for the application.

Appendix A: Selected Portions of Model