# Gedae: Auto Coding to a Virtual Machine


| William I. Lundgren | Kerry B. Barnes | James W. Steed |
|---|---|---|
| Gedae, Inc. | Gedae, Inc. | Gedae, Inc. |
| 18000 Horizon Way | 18000 Horizon Way | 18000 Horizon Way |
| Suite 200 | Suite 200 | Suite 200 |
| Mt. Laurel, NJ 08057 USA | Mt. Laurel, NJ 08057 USA | Mt. Laurel, NJ 08057 USA |
| (856) 231-4458 | (856) 231-4458 | (856) 231-4458 |
| bill.lundgren@gedae.com | kerry.barnes@gedae.com | jim.steed@gedae.com |



## *Abstract*

Gedae is an integrated application development environment. It has been under development since 1987 – though the concepts involved are rooted in much earlier work done in the area of data flow. Gedae uses the idea of data flow, which is a statement of data requirements on ports to a functional unit, but largely discards the notion of building a behavior analogous to the flow of electricity or fluids.

In Gedae we have jointly developed a programming language, a virtual machine and transformations that map the program to the virtual machine. Together these components provide the capability to describe the application functionality independent of the architecture (the language) and automatically generate an efficient implementation (the transformations and virtual machine). In this paper, we discuss three primary topics – the language, virtual machine and transformations. Since a detailed discussion of each item is beyond the scope of this paper, we illustrate parts of each topic using a series of examples.


## *Introduction*

The language has two major requirements: the ability to directly express functionality and the ability to transform the functional specification into an implementation on a virtual machine. Gedae has been able to satisfy these requirements because the virtual machine has been designed alongside the language. The Gedae Language consists of function boxes that process data with input and output data ports. The ports have defined characteristics such as data type, token type, token requirements and behavioral descriptors. The language to express these characteristics has over 50 features. These features are designed to enable the direct specification of the important functional characteristics of signal and data processing algorithms, distribution including load balancing and fault tolerance, and application (alternatively software or mode) control.

The virtual machine consists of a runtime kernel provided by Gedae. The kernel has a command handler, a static execution engine, a dynamic scheduler and various other support components. The application consists of a mix of library and developer provided functions along with components generated by Gedae, such as, static schedules for preordered execution and dynamic schedules for execution sequences that vary at runtime. The virtual machine also allows for vendor specific optimizations of processing, such as, setting data transfer parameters. One of the unique features of Gedae is the visibility of the implementation and execution it provides. This visibility is possible because the language (that is the block diagram), the transformations and the virtual machine are all part of Gedae. This paper describes the virtual machine, and particularly, the visibility of the implementation and execution.

Over 100 complex algorithms are used to implement the transformation from the block diagram to the implementation. In this paper, we give examples of some of the language features, and where appropriate, the transformations that provide highly efficient implementations and correct implementations. The first example illustrates the simplest of all port characteristics – static data requirements. The second example shows data requirements that change during the course of execution. The third is the insertion of glue code necessary to invoke the signal and data processing algorithms on an embedded processor. The fourth example is the user's ability to tune the static scheduling to significantly save memory in a SONAR

processing chain. The fifth example is memory sharing among processing modes of a multimode RADAR. Finally, we briefly describe ongoing work to extend the virtual machine to include hardware components, such as, FPGAs in addition to the DSP processors that are the focus of this presentation.

## *RADAR Pulse Compression Illustrates Static Data Requirements*

This section illustrates static data requirements on IOs of function boxes. The Gedae block diagram (see Figure 1) implements RADAR pulse compression.

The downstream processing uses an FFT, therefore, x_vx turns the scalar data into a vector of length N as indicated by the [N] in line 1. N scalars are needed to form the vector. The input

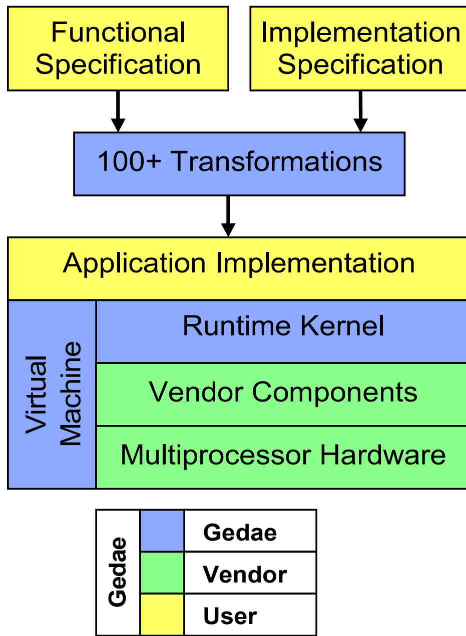


**Figure 1 - The Structure of Gedae**

imposes a data requirement of N input tokens for each output vector as indicated by the (N). Note that the size of the vector (also the input data requirement) is the integer input parameter specified in line 2. The parameter input does not have the stream modifier.

The box vx_zfill uses the parameter input Pad (line 8) to add to the length of the vector. The vector length of out equals N (the vector length of in) plus Pad. The remaining stream inputs and outputs all specify one vector of length N. Line 5 specifies a vector parameter of length N.

The data requirements in this example do not change while running, therefore, the execution can be preplanned in a static schedule to provide an efficient implementation.

| | | **x_vx** |
|---|---|---|
| 1 | Inputs | stream complex in(N); |
| 2 | | int N; |
| 3 | Output | stream complex out[N]; |
| | | **vx_multV** |
| 4 | Inputs | stream complex in[N]; |
| 5 | | float V[N]; |
| 6 | Output | stream complex out[N]; |
| | | **vx_zfill** |
| 7 | Input | stream complex in[N]; |
| 8 | | int Pad; |
| 9 | Output | stream complex out[N+Pad]; |
| | | **vx_fft** |
| A | Input | stream complex in[N]; |
| B | Output | stream complex out[N]; |

**Table 1 - Function Box IO Data Requirements**

## *Mode Control for Radar Illustrates Dynamic Data Requirements*

Mode control is inherently dynamic as mode changes are not preplanned but occur due to changes in the environment. During part of the operation, data is being processed through one part (mode) of the graph and not through others. This processing means that the data requirements at any given time will depend on the mode of operation. The IO data requirements are shown in Table 2, and the top-level flow graph is shown in Figure 2.

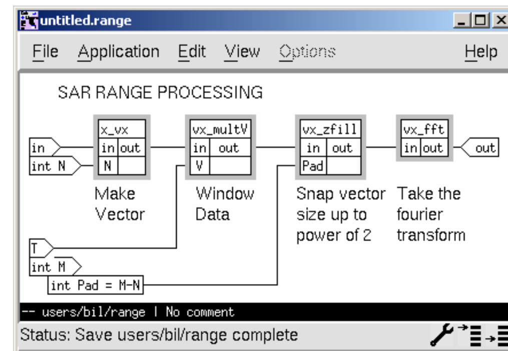


**Figure 2 – Gedae Block Diagram for Pulse Compression**

| mx_segbranch_c | | |
|---|---|---|
| 1 | Input | stream complex in[R][C]; |
| 2 | | stream int c; |
| 3 | Output | segmented dynamic stream complex out0[R][C]; |
| 4 | | segmented dynamic stream complex out1[R][C]; |
| **mx_merge_c** | | |
| 4 | Input | nondet stream complex in0[R][C]; |
| 5 | | nondet stream complex in1[R][C]; |
| 6 | | stream int c; |
| 7 | Output | stream complex out[N]; |

**Table 2 - Function Box IO Data Requirements - Mode Example**

The box mx_segbranch_c distributes the complex input matrix to the appropriate output based on the value of the input control token c. In Figure 2, if all data is sent to the branch for ModeA, then only the subgraph that implements the mode will execute. The modifier dynamic indicates that, during execution, the box is responsible for informing the Gedae Run Time Kernel (RTK) of the actual number of tokens (matrices) that were placed on the output. The modifier segmented indicates that the box will place a segment marker on the output when the downstream graph should be reset. A reset causes the software to be brought back to a known state so that a new mode can be started.

The box mx_merge_c reassembles a stream of data after the mode subgraphs (ModeA and ModeB in Figure 3) have completed the mode processing. The modifier nondet indicates that the box is responsible for telling the Gedae RTK how many tokens are required. Effectively the box runs in two phases – first it tells the Gedae RTK how many tokens are required on each input based on the values of the control tokens c.

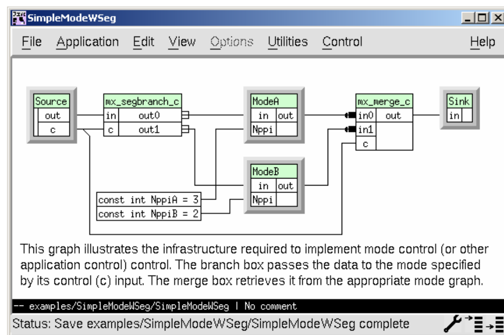


**Figure 3 - Mode control using a branch / merge pair**

For example, if c has three tokens, a 0, 1 and 0, then the box will tell the Gedae RTK it requires two tokens on in0 and one token on in1. Later, when the tokens are available, Gedae will invoke that box so it can complete the processing.

The graph in Figure 2 directly expresses mode control. Using typical development methodologies, the complexity of implementing mode control for large distributed systems can be high Gedae implements it automatically.

## *Gedae Transformation to Implement Distribution Illustrated using SAR Processing*

In SAR processing, the processing of rows and columns of a matrix is distributed across multiple processors. After the row processing is complete, the data has to be redistributed so that the columns can be processed. This redistribution is often called a distributed corner turn. The graph in figure 3 implements a distributed corner turn. The r is a range variable (elsewhere known as an iteration variable) that specifies the number of processors the row processing will be distributed across. The c is the range variable for the column processors. The black shadow indicates that there are multiple boxes.

Figure 4 shows the actual boxes for a graph with both r and c equal to {0,1}. The boxes with the green shading go on one processor and those with the yellow shading go on a second processor.

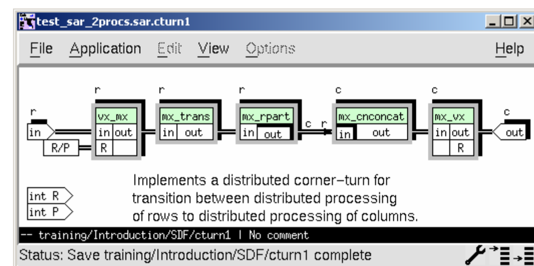


**Figure 4 - Gedae Block Diagram of a Distributed Corner Turn**

The green squares shown on the outputs of the mx_rpart boxes and the inputs of the mx_cnconcat boxes are send / receive pairs inserted by Gedae to implement the data movement. The red receive on [1]mx_cnconcat indicates that at the moment this snapshot was taken that receive was waiting for data. Deadlock can occur when the transfers are blocking, that is, the sender must wait for the receiver to accept the data before it can continue execution. In the above example, if both processors are sending data and neither can receive the data from the other until the send is complete, then no progress

can be made. The deadlock that might occur is classic. Gedae's transformations analyze for this condition and schedule the execution of one processor to send (just a function box inserted in the block diagram) followed receive and the other to receive followed by send. Notice that this point the block diagram has been extended to include an ordering beyond the ordering.

## *RADAR with Mode Control and Memory Sharing*

In the example shown in Figure 2, there are two modes of operation. These modes each require separate memory resources to process the data. If the modes operate independently, then there is no reason to require separate buffers. As a result we have extended the Gedae language to provide information so that Gedae can allocate one block of memory that is shared by a set of modes.

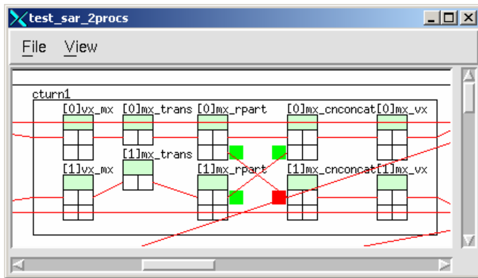


**Figure 5 - Flattened Graph with Send / Receive Pairs**

The graph is exactly the same but mx_segbranch_c is replaced with mx_excbranch_c. The exc prefix is an abbreviation for exclusive. The only difference in the IOs is the addition of the modifier exclusive to the outputs. This addition tells Gedae that the two modes will never execute at the same time, so the shared memory will be reset to a known state at the beginning of each segment of processing. It can't possibly interfere with the processing of the other mode because it will be reset before the commencement of mode processing in that segment. Gedae automatically implements this memory sharing. One customer's application shows an 80% savings in memory use.

## *SONAR with Subscheduling and Memory Sharing*

Figure 6 is the block diagram for a SONAR processing chain. The characteristic of this type of processing that lends itself to memory optimizations like strip mining is a great decimation in the data rate as the data proceeds through the processing chain. The example shown in Figure 6 has the decimations shown in Table 3. These decimations vary from implementation to implementation but are representative of the sorts of decimations one can expect.

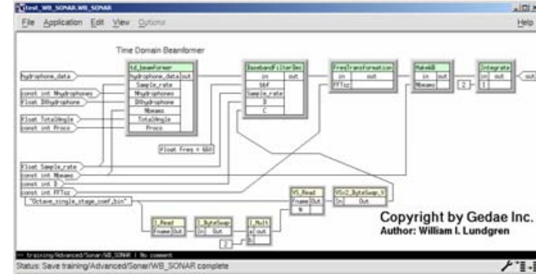


**Figure 6 - Gedae Top Level Block Diagram of Wideband SONAR Processing**

The total decimation of 4096 means that 4096 vectors (1 element from each hydrophone) are needed to produce a single output. In this example we used 64 hydrophones. The schedules require 258,260,680 bytes of memory.

| Function Box Name | Decimation |
|---|---|
| td_beamformer | 1 |
| BasebandFilterDec | 4 |
| FreqTransformation | 1024/240 |
| MakeWB | 120 |
| Integrate | 2 |
| Total Decimation | 4096 |

**Table 3 - The Decimations for Components of the SONAR Processing Chain**

Fortunately, we can process one input vector at a time until four are available for the next stage. We repeat that process until there are 1024 tokens for FreqTransformation (the word transformation in this name refers to one of the operations in the SONAR processing chain and should not be confused with the Gedae transformations). Once we have those tokens, we can produce 240 outputs and so on. When Gedae sets up the processing using what we call subscheduling (one use of Gedae subscheduling is to implement strip mining), the memory requirements shrink to 2,778,912. To subschedule an application, Gedae applies subscheduling transforming implementation, so a smaller schedule executes multiple times.

While the savings in this example are significant, a customer saw a much greater savings. Their

real SONAR problem required 47 Gbytes before sub-scheduling and 40 Mbytes after.

## *Image Processing with Fine Grained Parallelism*

Increasingly, field programmable gate arrays (FPGAs) are being used alongside DSPs as a method for meeting the throughput and latency requirements of real time systems. FPGAs allow developers to implement applications in hardware where many operations in an algorithm happen concurrently. For example, image kernel applications, such as edge detection, can be constructed so that each multiply and add execute in parallel, allowing pixels to be processed at a very high rate. To support such fine grain parallelism, Gedae has been augmented with a single sample graphical meta-language based on the theory of register transfer languages called Gedae-RTL. This single-sample addition to the language allows Gedae to manage each operation of the algorithm independently.

In the image kernel example in Figure 7, each multiply and add is specified as a separate component in the graph, unlike a for-loop in C-code, which hides the parallelism of the algorithm in a single component.. Gedae is able to analyze this netlist of algorithmic components and transform the specification to optimize execution, for example, adding delay registers to ensure the result of each stage settles in an FPGA pipeline. The Gedae-RTL graph outputs target-optimized code in any language via a Language Support Package, whether that be specialized C-code for an AltiVec or VHDL-code for an FPGA.

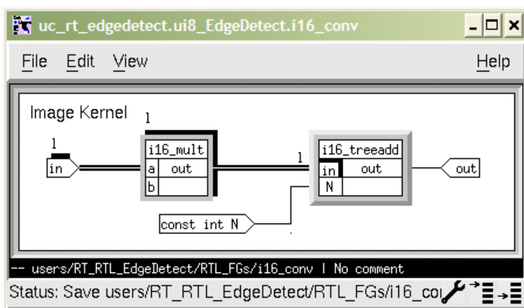


**Figure 7 - Gedae-RTL Image Kernel Operation**

The image kernel operation in this example is a small part of a larger system mapped to the FPGA. The graph shown in Figure 8 adds a real time data source (camera) and sink (VGA monitor). In addition, frames are sent to the host to recalculate the threshold (also implemented in Gedae), and the adapted thresholds are received by the FPGA and applied to the data in future frames.

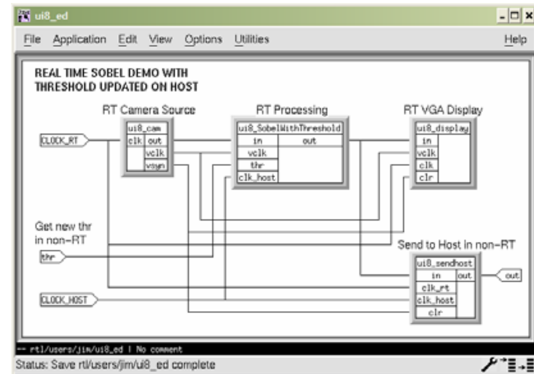


**Figure 8 - Real Time Sobel System with Adaptive Thresholding**

## *Note on Computational Efficiency of Gedae Applications*

The implementations are efficient. The paper listed in the bibliography entitled "Gedae Runtime Kernel Performance Characterization" by Kerry Barnes measures the percentage time spent by the infrastructure relative to the time spent in signal processing operations like FFTs and vector multiplies. It shows that for Altivec processors, even if all the Gedae infrastructure could be removed, the savings in computation time decreased by, at the most, 10% and in some cases by, as little as, 1%. Of course, in hand coded applications, the infrastructure must be implemented by hand and generally dominates the code.

## *Conclusions*

The unique characteristic of Gedae is the close integrated language, transformations and virtual machine. The language is rich and provides direct expression of any functionality. Gedae transformations automatically build an application that executes efficiently on heterogeneous multi-processor hardware. This paper illustrates several of the more than 100 transformations.

## *Bibliography*

Peterson, J., Petri Net Theory and the Modeling of Systems, Prentice-Hall, Englewood Cliffs, NJ, pp. 214-221, 1981.

E. A. Lee and D. G. Messerschmitt. Synchronous Dataflow. Proceedings of the IEEE, 75(9):1235-1245, September 1987.

Richard M. Karp and Raymond E. Miller. Properties of a Model for Parallel Computations, Determinacy, Termination, and Queueing. SIAM Journal of Applied Mathematics, 14(6):1390--1411, November 1966.

Ackerman, W.B., "Dataflow Languages," COMPUTER, Vol. 15, February 1982.

Barnes, K.B., et al., "A Data Flow Graph Programming Environment for Embedded Multiprocessing", Technical Report, GE Advanced Technology Laboratories, Moorestown, New Jersey, 1992.

D. Kaplan and R. Stevens. Processing Graph Method 2.0 Specification. Technical report, Naval Research Laboratory, September 1995.

Barnes, K. B., “Gedae Runtime Kernel Performance Characterization”, Gedae Users’ Conference I, March 2003.