# GEDAE: A TOOL FOR IMPLEMENTING SOFTWARE RADIO ON HETEROGENEOUS SYSTEMS

James Steed (Gedae, Inc., Mount Laurel, NJ; jim@gedae.com),
William Lundgren (Gedae, Inc. Mount Laurel, NJ; bill@gedae.com),
Kerry Barnes (Gedae, Inc., Mount Laurel, NJ; kerry@gedae.com)

## ABSTRACT

This paper discusses recent language extensions to the Gedae programming environment. The first language extension allows application developers to more easily develop modal software in Gedae's data flow programming language. By breaking Gedae's infinite data streams into finite length segments, mode changes become natural parts of the languages, implemented as side effects on segment boundaries. The second language extension expands the range of targets Gedae can support. Developed to support firmware targets such as FPGAs, Gedae-RTL allows developers to specify algorithms using a graphical single sample language and export that specification into code in a target language such as VHDL. Although developed for VHDL and FPGAs, the Gedae-RTL capability is generic enough that any language can be targeted. With these two language extensions, Gedae developers can more easily develop full modal software radio systems and port them to heterogeneous targets.

## 1. INTRODUCTION

In order to achieve the throughput and latency requirements of many software radio (SWR) applications, multiple processors must be used. Gedae is an integrated design environment for deployed systems and advanced demonstrators based on boards of digital signal processors (DSP) (e.g., AltiVec, PowerPC, TigerSHARC) or distributed networks (e.g., Linux clusters). Its rich block diagram language streamlines and simplifies the task of building applications for distributed systems. The block diagram provides a highly compartmentalized depiction of the algorithm, suitable for partitioning. This block diagram created by the developer specifies only the functionality of the graph, without regard to the target system. Under the direction and control of the user, Gedae is able to use its knowledge of the target (e.g., its processor layout, transfer methods, and optimized routines) to transform the graph into an efficient implementation of the application on the target processors.

Several language features make Gedae particularly powerful for SWR applications. For example, data streams are easily specified in Gedae, and the language allows developers to mark segments of streams. These developer-specified markers on the beginning and end of stream segments can produce side effects that affect graph behavior. An excellent example of such a change in graph behavior is a mode change in a SWR application. Gedae also has a full suite of analysis tools for observing and debugging execution on the host and DSPs, such as the Trace Table where all execution, transfers, and mode changes are displayed.

Increasingly, field programmable gate arrays (FPGAs) are being used alongside DSPs as a method for meeting these data flow requirements. These FPGAs are often used to implement front-end signal processing that must be processed at a high throughput. With the increased focus on targets such as FPGAs, the Gedae block diagram language has recently been extended to also enable porting to firmware. Unlike the AltiVec, PowerPC, and TigerSHARC these new targets generally do not allow cross-compilation of C-code. To support other languages, Gedae has been augmented with a single sample graphical meta-language based on the theory of register transfer languages called Gedae-RTL. This language is capable of exporting VHDL code for FPGAs as well as Ansi-C code optimized for a DSP. Much like Gedae's core language, the Gedae-RTL graph specifies only the functionality of the graph without regard to the target or its programming language. Through Gedae's knowledge of the target processor, the graph is transformed to generate correct results on the target and for optimized performance on the target. Then target code is exported to implement the application. Components implemented in Gedae-RTL interact seamlessly with core Gedae components, allowing an entire heterogeneous system to be specified in the Gedae programming environment.

## 2. MODAL SOFTWARE

Gedae's language is based on data flow. A flow graph implements an application, and each primitive node in the flow graph defines the data flow relationship between its inputs and outputs. The three core types of data flow relationships are

- Static: the number of tokens produced and consumed is constant and determined at application start-up.
- Dynamic: the number of tokens produced and consumed is determined at runtime, and the node cannot execute unless full input queues are ready to be processed and empty output queues are ready to be written to.
- Nondeterministic: the number of tokens produced and consumed is determined at runtime, and there are no restrictions on when the node can execute.

While these basic types of data flow are sufficient to implement any application, complex modal applications would require large amounts of application control to be implemented in an ad hoc manner alongside the signal and data processing. To reduce this overhead and provide a general solution to the problem of modal software development, the Gedae language has been extended to allow developers to mark segments of streams. These user-specified markers on the beginning and end of stream segments can produce side effects that alter graph behavior, such as switching to tracking mode after a target has been found in a stream of radar data.

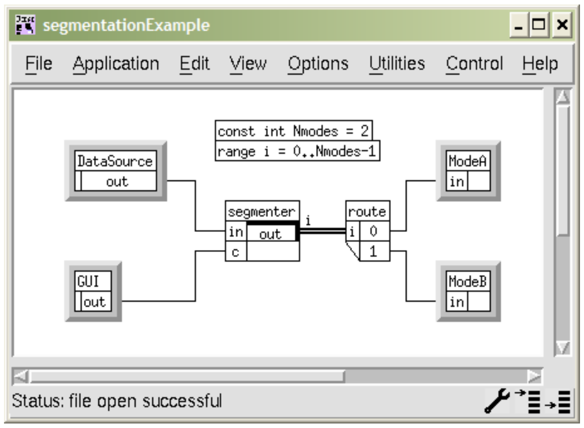


**Figure 1 - Two-mode application implemented using segmentation**

Gedae's primitive language is based on C with functional and variable-based extensions to allow the developer to interface with Gedae's data structures. This C-code is grouped into methods, e.g., the `Start` method is executed at start-up, the `Apply` method is executed when the primitive has data to process, etc. The example two-mode application is shown in Figure 1. Two subgraphs implement the two modes, `ModeA` and `ModeB`. The `segmenter` creates two branches of data and uses the `segment()` function to place the segment markers on the two streams. As the markers are encountered in downstream primitives, the `Reset` and `EndOfSegment` methods are invoked, creating side effects and forming distinct boundaries between modes.

### 2.1. Segmentation

To implement `segmenter` primitive, only two language features are needed on top of Gedae's standard primitive language: the ability to mark streams as segmented and the ability to place markers on the segmented streams. The input/output list for a Gedae primitive specifies the data type, token type, data flow parameters, and name of all the primitive's inputs and outputs. Streams are declared using the `stream` modifier. For example, the input list to the `segmenter` primitive is declared as

```
Input: {
  stream float in;
  stream int c;
}
```

The output out to the `segmenter` primitive groups together two output streams in a family, as indicated by the shadowed border to the output in Figure 1 as well as the index `i` beside the output port. Both outputs are segmented streams so their declaration marks them as such:

```
Output: {
  segmented dynamic stream
          float [N]out;
}
```

The pre-index `[N]` indicates the output is a set of `N` streams (or family of size `N`), and the modifier `segmented` marks the outputs as segmented streams. Thus, to declare an output stream as segmented, the developer only needs to add the modifier `segmented` to the declaration. This example segments a dynamic stream. Mode control is inherently dynamic or nondeterministic 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. (In other words, a static output stream cannot be segmented.)

Marking the beginning and end of segments is just as natural. The `Apply` method processes data in queues. On each execution of the `Apply` method, the `segmenter` primtive loops through the input queue and copies tokens from the input to one of the output queues according to the control tokens in the `c` queue (or drops the token, if desired). Inside that loop, the developer can put begin and end of segment markers on segmented output streams by using the `segment()` function. In most examples, the developer only need place the end marker, as it is assumed that the first token produced after an end marker is the beginning of a new segment (the one counterexample where the begin marker is required is the case of zero-length segments). Thus, the for-loop inside `segmenter`'s `Apply` method is

```
for (i=0; i<size(in); i++) {
  if (c[i] != last) {
    segment(out[last],
            SEGMENT_END)
    produce(out[last],n[last]);
    n[last] = 0;
    last = c[i];
  }
  out[last][n[last]++] = in[i];
}
```

### 2.2. Exclusivity

Often in mode-based applications, the modes execute exclusively, that is only one mode is actively processing data at any given time. When a family of segmented streams is used to implement mode control, as in the example above, the streams can be declared as exclusive. An exclusive output allows Gedae to transform the implementation such that the modes share resources such as memory. As before, the declaration only requires an extra modifier in the output list,

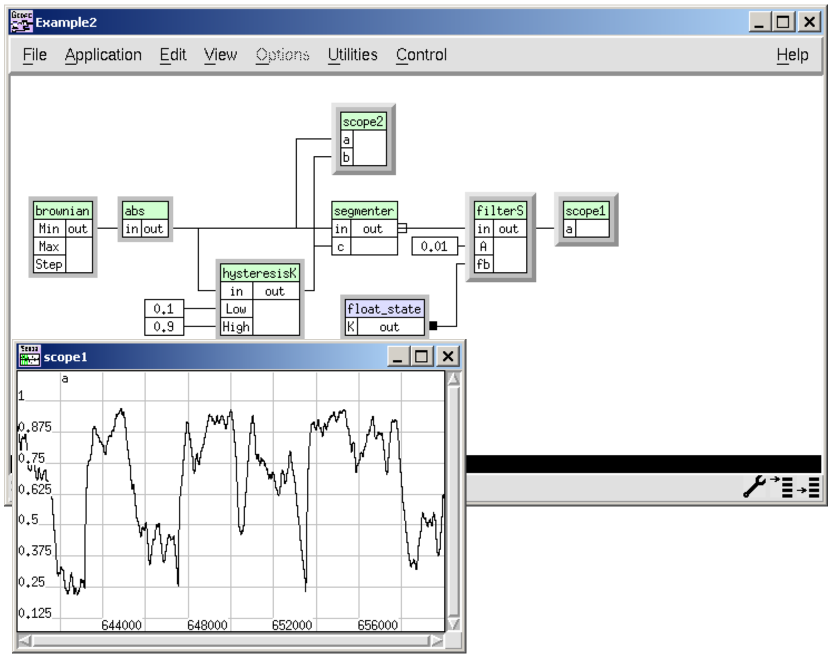


**Figure 2 - External state allows retaining state information between segments**

```
Output: {
  exclusive segmented dynamic
         stream float [N]out;
}
```

No other programming is required to utilize the resource sharing.

### 2.3. External State

One of the essential features of segmentation is that state information is cleared and reset at the segment boundary, allowing for the execution of the next instance of the mode to begin just as if it was the first instance. While this behavior is fundamental to many applications of segmentation, there are also examples where state must be retained across segment boundaries. For example, a simple low-pass filter retains the last output token for use in computing the next output token. This last token is stored in the state vector of the primitive. If the token is stored in the internal state of the primitive, it will be cleared at segment boundaries.

To support this retention of state information across segment boundaries, external state variables can be declared. In Figure 2, the `float_state` primitive provides an external state variable that is shared between segments in the `filterS` subgraph (which implements this low-pass filter). The scope shows

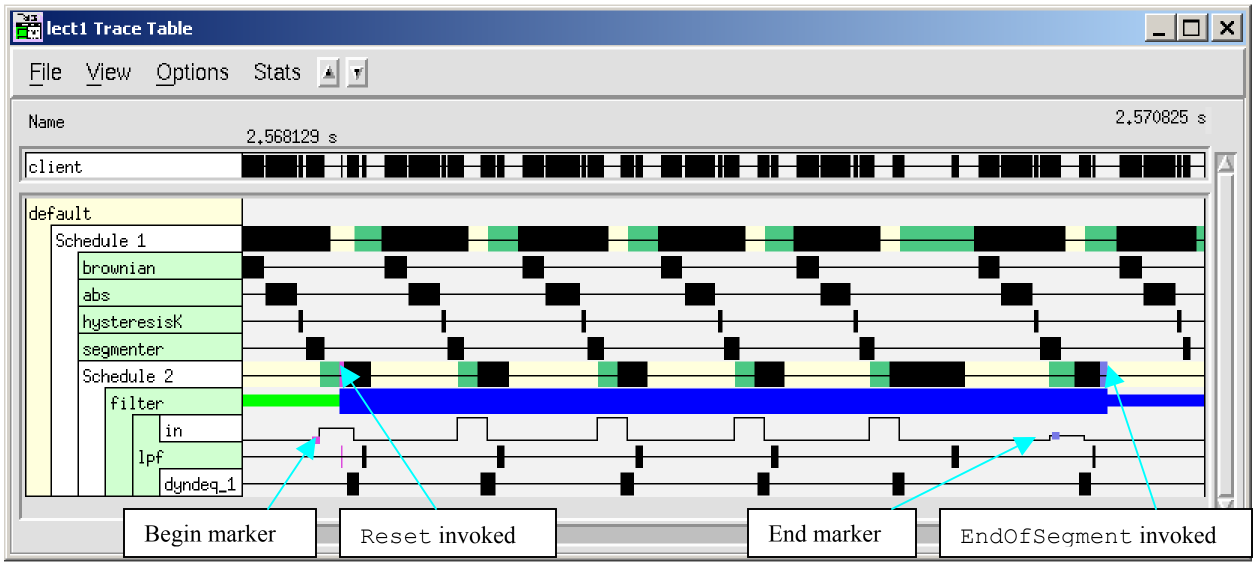


**Figure 3 - The Trace Table provides visibility to all events, including segmentation**

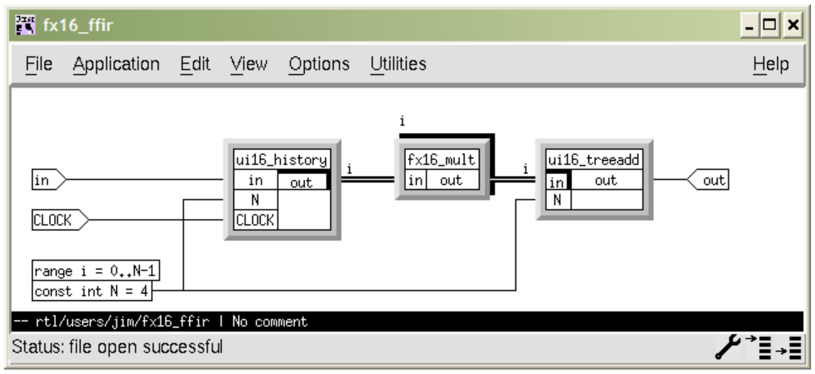
correct data with smooth transition between segments. Segment markers in the stream do not affect the value of external state variables. These variables can also be shared between modes so that all modes share some of the same state information.

### 2.4. Visibility

Gedae's suite of analysis tools has been extended to support the new segmentation language features. Gedae displays timing information in the Trace Table, allowing developers to see precisely where each box executed, including boxes inserted by Gedae like sends and receives, as well as analyze processor load. The Trace Table has been extended to also display segmentation related events. Figure 3 shows some of these events. The small magenta box on the line beside the `in` input to the `lpf` primitive indicates a begin marker to a stream segment. This begin marker causes the mode to reset by executing all the `Reset` methods of all the primitives in the schedules for this mode. In this example, the short magenta area on the line beside "Schedule 2" shows the Reset methods of the primitives in this mode resetting. Similarly, the small gray box near the end of the `in` event line represents an end marker to the stream segment. The short gray bar on the "Schedule 2" event line shows the `EndOfSegment` methods of the primitives executing. Similarly detailed information has been added to the flow graph editor and the flattened view of the graph to aid debugging and analysis.

## 3. FIRMWARE TARGETS

In embedded systems, FPGAs are often used alongside DSPs to implement front-end signal processing that must be processed at a high throughput. With the increased focus on targets such as FPGAs, the Gedae block diagram language has been extended to enable porting to firmware. Unlike the AltiVec, PowerPC, and TigerSHARC, these new targets generally do not allow cross-compilation of C-code. To support other languages, Gedae has been augmented with a single sample meta-language based on the theory of register transfer languages called Gedae-RTL. This language is capable of exporting VHDL code for FPGAs as well as Ansi-C code optimized for a DSP.



```
range i = 0..N-1
const int N = 4
-- rtl/users/jim/fx16_ffir | No comment
Status: file open successful
```

**Figure 4 – FIR filter implemented in Gedae-RTL using 16-bit fixed-point arithmetic**

Functionality built using Gedae-RTL uses the new single sample primitive type. Conceptually, a graph of single sample primitives forms a processing pipeline that is enabled by a clock. These single sample primitives are built upon seven fundamental functions: register, assignment, decimate, clock, memory, memory read, and memory write.

The register function (`R(in,out,clk)`) copies the input variable to the output with a delay of one clock pulse. A register stores the state information in a graph. When a Gedae-RTL graph is reset, the registers return to their initial value.

The assignment function (`A(expr,out)`) evaluates an expression and assigns its value to a variable. For example, an addition primitive may be programmed with the line `A(a+b,out)`. To allow more flexibility in programming single sample primitives, the function function (`F(in0,in1,…)(out0,out1,…){lines_of_code}`) extends the assignment function by allowing the developer to put multiple lines in a single function.

The memory function (`M(out,len,bytes)`) declares a memory buffer. The definition of memory buffer here is very generic; it could be a QDR SRAM bank attached to the FPGA, an output pin with an address space of 1 bit, a block RAM generated on the FPGA, etc. The memory read (`MR(addr,out)`) and write functions (`MW(in,addr)`) access a buffer declared by the M-function.

The decimate (`D(in,clk)`) function ties a clock to a variable. It is different from the R-function in that it does not induce a delay (there is no output variable) and it does cause the variable to be static. The clock (`C(in,clk)`) function retrieves the clock tied to a variable.

### 3.1. Language Independence

Much like Gedae's core language, the Gedae-RTL graph specifies only the functionality of the graph without regard to the target or its programming language. For example, Figure 4 shows a FIR filter implemented in Gedae-RTL, built from a register pipeline (`ui16_history`), multipliers (`fx16_mult`), and a tree-adder (`ui16_treeadd`). The graph in Figure 4 includes only multipliers, adders, and delay registers. In other words, the graph is not specific to FPGAs or firmware; it is simply a specification of a FIR filter. Gedae is able to export code in potentially any language to implement the functionality specified in the graph.

To support any language, the Language Support Package (LSP) has been integrated into the Gedae-RTL code generation process. This process is shown in Figure 5. The Gedae-RTL graph is not translated directly into code. Instead it is transformed into an internal netlist representation and information on the algorithm is collated in data structures. Implementation settings from the graph developer also affect the internal data structure, allowing the developer to insert registers and map memories to hardware components to enhance the implementation. Once the internal implementation is created, the LSP uses the information provided by Gedae to export code in the target language. The target language could be VHDL for an FPGA, assembly code for another alternate architecture, or Ansi-C code with DSP-specific enhancements. The code is then built by Gedae's customizable make system.

### 3.2. The Use of Cores

In FPGA and firmware programming, developers often utilize processing cores – pre-constructed components whose efficiency and synchronization issues have already been addressed. The language independence of a Gedae-RTL graph does not

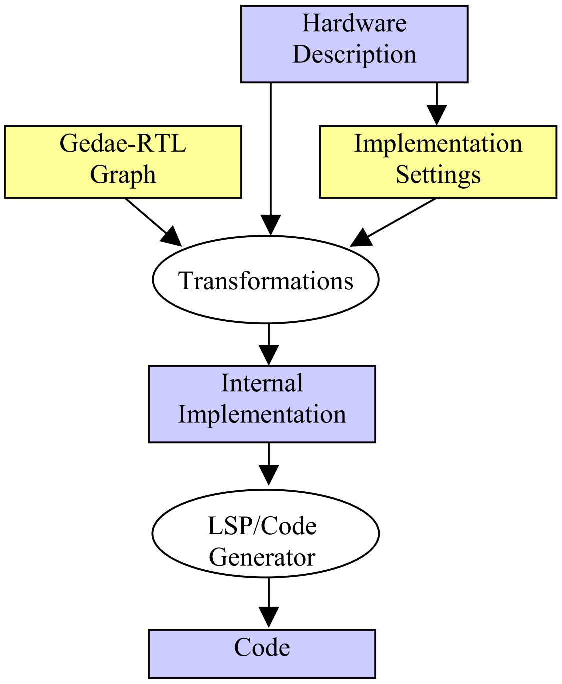


**Figure 5 - Framework for code generation**

hamper the use of cores. Primitives in a Gedae-RTL graph can provide two implementations: a generic one that works on all processors and a target-specific one. The `Function` method is used to specify the generic implementation of a Gedae-RTL primitive. If a pre-optimized core is available to implement the same function, the code to utilize this core can be placed in the `Core` method of the primitive.

### 3.3. Heterogeneity

Components implemented in Gedae-RTL interact seamlessly with other Gedae components in the flow graph, allowing an entire heterogeneous system to be specified in the Gedae programming environment. If the Gedae-RTL component is mapped to an FPGA or other firmware target, the Gedae make system will output a binary image, and Gedae will load the FPGA with that image. The DSPs are simply running schedules of other primitives in the Gedae graph. When the DSPs fill the FPGA's input buffers, it will signal to the FPGA to process the data. When the FPGA has filled its output buffers, it will signal to the DSPs when its output buffers are full. The segmentation features described earlier can also drive Gedae-RTL-created components. For example, when a mode resets, the Gedae-RTL-specified FPGA also resets, clearing the state in its registers, and reinterpreting its input parameters to determine its functionality in the new mode.

## 4. CONCLUSION

With the segmentation and Gedae-RTL language extensions, application developers can use Gedae to easily program modal software radio systems and can map those systems to heterogeneous architectures. Segmentation allows developers to mark segment boundaries on streams. These markers cause side effects downstream, triggering execution of the `Reset` and `EndOfSegment` method, providing a natural way to initialize software at the start of modes and summarize results at the end of modes. Processing can also be specified in the Gedae-RTL single sample language, and these graphs are target language independent. The LSP allows a Gedae-RTL graph to be translated to any firmware or software language, including VHDL to implement a FPGA.