Implementing Modal Software in Data Flow for Heterogeneous Architectures

James Steed, *Kerry Barnes*, and *William Lundgren*
Gedae, Inc.,
Phone: 856-231-4458
Email Address: {jim,kerry,bill}@gedae.com

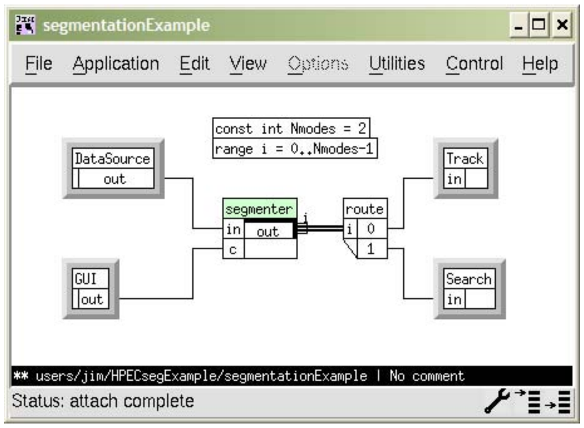
Software for embedded systems is often based on distinct processing **modes**. A simple example of such modal behavior is a radar system that switches between search mode and tracking mode as targets are located. In complex software systems, the system may have dozens of modes, including sub-modes, forming a deep hierarchy. Such large embedded systems often must be implemented on boards of multiple digital signal processors (DSP). Increasingly, field programmable gate arrays (**FPGA**) are being used alongside DSPs as a method for meeting the throughput and latency requirements of these systems. Gedae is an **integrated design environment** for deployed systems and advanced demonstrators based on DSPs (e.g., AltiVec, PowerPC, TigerSHARC) or distributed networks (e.g., Linux clusters). This paper describes extensions to Gedae's language that empower developers to easily develop modal software and enable them to port that software to **heterogeneous architectures**, including a new class of boards that contain both DSPs and FPGAs.

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.

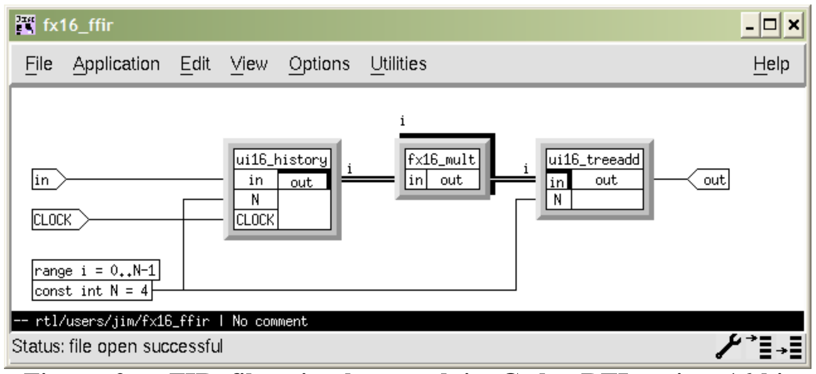
While these basic types of data flow are sufficient to implement any application, complex modal applications would require large amounts of application control to be implemented in an ad hoc manner alongside the signal and data processing. To reduce this overhead and provide a general solution to the 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.



```
const int Nmodes = 2
range i = 0..Nmodes-1
```

** users/jim/HPECsegExample/segmentationExample | No comment
Status: attach complete

Figure 1 – Two-mode radar implemented using segmentation



-- rtl/users/jim/fx16_ffir | No comment

Status: file open successful

Figure 2 – FIR filter implemented in Gedae-RTL using 16-bit fixed-point arithmetic

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 radar application is shown in Figure 1. Two subgraphs implement the two modes, `Track` and `Search`. The `segmenter` primitive reads data from an I/O device (`DataSource`) and a graphical user interface (`GUI`) to create two branches of data and uses the `segment()` function to place the segment markers in the 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.

Heterogeneity

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.

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 copies the input variable to the output with a delay of one clock pulse. The assignment evaluates an expression and assigns its value to a variable. The memory function declares a memory buffer, and the memory read and write functions access a buffer. Decimate and clock functions set and retrieve the clocks tied to variables.

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 2 shows a FIR filter implemented in Gedae-RTL, built from a register pipeline (`ui16_history`), multipliers (`fx16_mult`), and a tree-adder (`ui16_treeadd`) with no target-specific processing. Through Gedae's knowledge of the target processor, a graph such as this FIR filter 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.