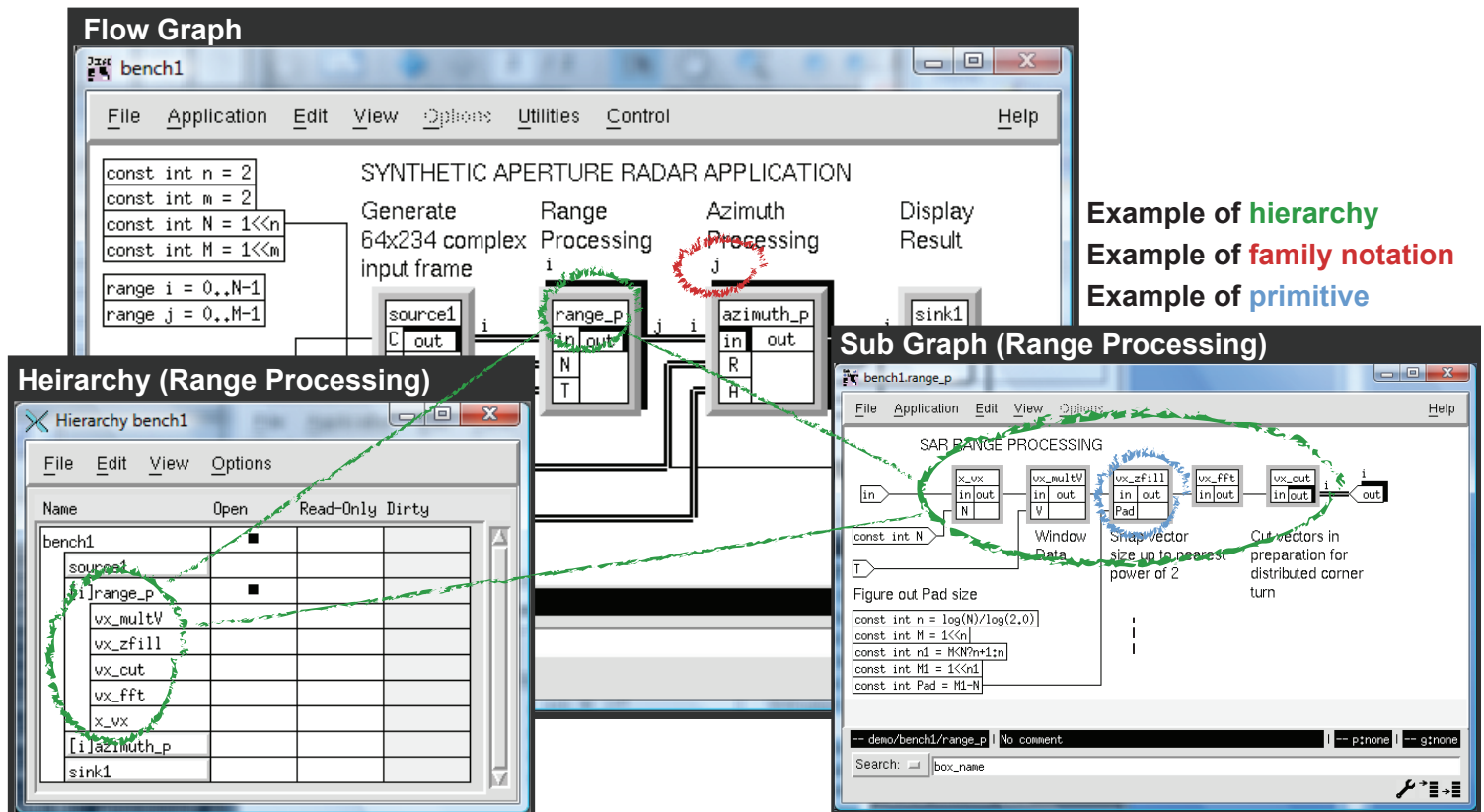


## Executive Summary

Gedae offers a powerful and fully expressive block diagram based language that provides simplicity of expression and platform independence by hiding the language details. The application can then be easily partitioned and mapped to the hardware of your choosing.



Gedae's language provides a powerful way of describing an application, expressing the algorithm while hiding the language details.

## Block Diagram Language

Gedae applications are designed using a block diagram language. The language uses two types of blocks: primitives and subgraphs. **Primitives** define pieces of code that perform operations. Subgraphs define a set of blocks (of both types) and the connections between the blocks. Gedae's block diagram method of expressing an application is a powerful form of abstraction that fully expresses the functionality while hiding the implementation details. The language also allows applications to be organized **hierarchically**, thereby promoting organized application design and code reuse.

The expressiveness of the language is enhanced with the ability to add text to the canvas of the graph to document the graph's operations. HTML can be used to enhance the appearance of the canvas text.

In addition, links to webpages, documents and external applications, such as requirement traceability tools, can be added to the text.

Another powerful component of the Gedae block diagram language is the **family notation**. While any Gedae flow graph can be partitioned and mapped to target hardware, families provide a direct way of expressing the distribution of data and processing. Any box on the canvas can be turned into a family of N boxes, and it is very natural to think of a family of N boxes being mapped to N processors.

Nontrivial distribution of data to a family of boxes is also easy to express using route boxes. Route boxes are used to express communication patterns between families of boxes using formulas.

## Partitioning and Mapping

One of the most powerful features of Gedae is the ease of partitioning and mapping an application to run on multiple processing elements (not to mention repartitioning and remapping). By creating an application as a flow graph, it is easy and natural in Gedae to partition the graph into sections, and then map each of those sections to hardware.

This act of partitioning the graph provides information to the Gedae compiler, which helps it plan the application's threads and adjust for the distribution the developer has specified.

When a graph is ready to be distributed, the developer must first partition the graph. In the Partition Table, the developer is presented with a table listing all the components in the flow graph. The developer simply selects which components should be broken off into a new partition and assign them to a new partition name.

Mapping those partitions to target hardware is just as easy. In the Map Partition Table, the developer is presented with a table listing all the partitions he has just made in the Partition Table. For each partition, the developer selects the processor number the partition will be mapped to from a predefined list.

Through selection of the optimal transfer method, ease of mapping does not lead to inefficiency in Gedae.

Name	Part	SubSched
[0]range_p	0	
[1]range_p	1	
[2]range_p	2	
[3]range_p	3	
[0]azimuth_p	0	
[1]azimuth_p	1	
[2]azimuth_p	2	
[3]azimuth_p	3	

Partition and map dialogs increase productivity by making porting the application the easiest step in the implementation.

### Mapping

Name	Proc Num	System Name	Trace Mem	Trace Size	Params
0	100	ent	default	1000	
1	101	ent	default	1000	
2	102	ent	default	1000	
3	103	ent	default	1000	

One would expect that an application so effortlessly mapped to hardware would perform basic, inefficient communication schemes, but Gedae is designed to not fall prey to such inefficiencies. Many transfer methods can be made available in Gedae, from DMA to shared memory to processor-specific protocols, and through the Transfer Table, the developer can easily select transfer methods for each communication. The transfer methods are fully parameterized, allowing for precise specification of buffer sizes and other parameters so that the most efficient transfer can be used.

### Transfer Methods Table

Name	Source	Dest	Xfer Type	NBSize	Send Bufs	Recv Bufs	Xfer Params
[0]azimuth_p.vx_mux<[1]in	101	100	stream_socket	0			
[0]azimuth_p.vx_mux<[2]in	102	100	stream_socket	0			
[0]azimuth_p.vx_mux<[3]in	103	100	stream_socket	0			
[1]azimuth_p.vx_mux<[0]in	100	101	stream_socket	0			
[1]azimuth_p.vx_mux<[2]in	102	101	stream_socket	0			
[1]azimuth_p.vx_mux<[3]in	103	101	stream_socket	0			
[2]azimuth_p.vx_mux<[0]in	100	102	stream_socket	0			
[2]azimuth_p.vx_mux<[1]in	101	102	stream_socket	0			
[2]azimuth_p.vx_mux<[3]in	103	102	stream_socket	0			
[3]azimuth_p.vx_mux<[0]in	100	103	stream_socket	0			
[3]azimuth_p.vx_mux<[1]in	101	103	stream_socket	0			
[3]azimuth_p.vx_mux<[2]in	102	103	stream_socket	0			