Code generation and the complexity of applications

By William I. Lundgren, President, Gedae, Inc.

One of the trends in computing today is to provide ever more capable hardware platforms for the implementation of modern applications. However, more capable hardware is often more complex to program. Because of this complexity, software development is a key problem and major risk that confounds developers of digital systems. Not only is the development difficult the first time the system is implemented, but also each subsequent evolution in the hardware requires restructuring and rewriting of the underlying software.

Software layers and autocoding, two methods for tackling the issues of new, complex platforms, have different ways of helping developers meet the requirements of modern applications. In comparing the two methods, the separation of functionality and implementation, (for example, the separation between the description of the algorithms and the coding needed to port the algorithms to target hardware) is a significant differentiator, and autocoding proves to be the best approach.

Software layers and autocoding

Both software layers, a way of grouping functionalities at the same level of abstraction, and autocoding, the automatic generation of software from a partitioned and mapped data flow graph, allow a developer to work at a higher level of abstraction – more in the application functionality and less in the hardware. In the case of software layers, the hardware-specific implementation is buried underneath a common, hardware-independent interface.

For example, the Message Passing Interface (MPI) is a software layer that provides a standard interface across platforms for inter-processor communications, and the Vector Signal Image Processing Library (VSIPL) is a software layer that provides a standard interface across platforms for hardware-optimized vector routines for signal processing. These approaches work extremely well in low-level layers such as socket communications, but as the layers become deeper, the interface moves further away from the hardware, and the software becomes more complex and less efficient.

Autocoding is fundamentally different, as it automatically constructs the implementation. A developer specifies the application in a simple, high-level, hardware-independent language (often graphic-based), and the autocoding tool translates this specification to lower-level, hardware-specific code. Autocoding often relies on software layers to provide some of the hardware independence, but its true value is in its ability to tailor parts of the implementation directly to the hardware. If the tool merely changes the graphical representation into code that utilizes software layers, it does not qualify as a true autocoder. Autocoders in their purest form utilize complex algorithms that have fine control over the implementation using low-level, efficient software layers.

Comparing the two methods

The use of software layers meets some programming needs for implementation at a higher level of abstraction, resulting in productivity improvement as well as some level of increased portability. On the downside, software layers must have the capability to respond to widely varying needs of applications, often making them too complex to be efficient. In contrast, code generation can generate more of the application while minimizing the complexity of the implemented software.

For example, consider how both approaches implement inter-processor communications. When memory management is needed to perform a transfer, some software layer systems provide libraries that incur the overhead of managing memory, even when the memory allocations could be made ahead of time. Separate libraries – those with and without memory management – could eliminate the unnecessary overhead, but put the burden on the developer to choose the correct library and preplan the memory allocation.

On the other hand, an analysis-based autocoding system manages the entire implementation – not just the communication layer. This type of system can first peruse the functional specification and preplan the inter-processor communications to eliminate the overhead. Figure 1 represents the functional specification of an application, and separately, the implementation on a multiprocessor system. In the case shown, the transfers are moving data from data memory on the source processor directly into data memory on the destination processor.

Separation of functional description and implementation detail

In either development approach, if the functional description of the application is cluttered with implementation details,

then portability will be limited. The key question is: What information is required in the functional description so that automatic implementation of efficient software is practical? While a paper description of a sensor system's functionality may be deemed sufficient information to generate an application, it does not provide practicality.

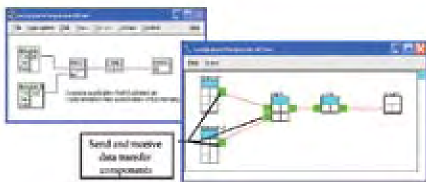

Figure 1

The focus has to be on providing information in a form that facilitates an implementation, while maximizing the quality of the implementation and minimizing the amount of implementation detail added to the functional description. While both software layers and autocoding provide this separation, autocoding provides a purer separation and better flexibility, which addresses the issues of software obsolescence and reduces implementation development time.

The addition of implementation to the functional description reduces portability because different architectures bring up different implementation issues. The implementation of an application on a shared memory architecture such as symmetric multiprocessing can be quite different from the implementation on a Direct Memory Access-based (DMA) architecture.

For example, in some systems, shared memory is not cacheable. In such a case, it is often important to get the data out of the shared memory at the first opportunity to avoid the execution inefficiencies. Although software layers can support such variations, either the layer is quite complex, giving rise to unnecessary overhead, or the developer must recognize the situation and make the necessary adjustments. With autocoding, the analysis and implementation routines can recognize the target-specific issues, and implement to accommodate and optimize for the various situations.

Analysis of implementation and reporting of runtime status

There is a wide range of code generation methods, but code generation has limited value if it is just the translation from one programming language to another with the insertion of large pieces of code. Instead, if there is an intermediate step in the code generation that forms an internal model of the software before outputting code, the model provides a method to analyze and optimize the generated implementation both more easily and in more fine detail. This level of analysis is unrealizable when using software layers alone.

Figure 2 illustrates the type of analysis that an autocoding tool can provide, given that it maintains an internal model of the software. Modern applications have dozens of components executing concurrently either as threads or on multiple processors, and the control necessary to operate these components can be very complex and can lead to difficult problems such as avoiding deadlock.

To illustrate one of these problems, Figure 2 creates a simple example with tokens dropped on one of the queues in order to artificially bring about a deadlock. In the block diagram of the functionality (bottom graph in the figure) data is being transferred down one path of processing that is calculating the average energy of the signal, and a primitive at the end of the branch is adjusting the average energy. In the implementation version of the block diagram (top graph in the figure), note the queues filled on one branch and starved on the other due to the dropping of tokens. The textual display describes the discovered problem, and the implementation display presents the information graphically. These capabilities are difficult to imagine in software libraries but are enabled by maintaining a software model in the development environment – something that is possible in an autocoding tool.

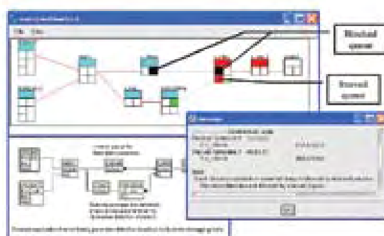

Figure 2

A winning solution for complex application development

A software tool that generates simple and efficient implementations can significantly reduce the issues associated with the growing complexity of application development. The issues associated with the practical separation of functionality and implementation need to be analyzed when generating code for complex hardware platforms. One autocoding tool, Gedae, is built based on the guiding principles to:

- Keep the functional specification free of implementation clutter
- Maintain a software model that is suitable for analysis and modification
- Autocode lean and efficient implementations

This type of tool helps developers more easily meet the requirements of modern applications.

For more information, visit the Gedae website at www.gedae.com.