

Model Driven Development – Measures of Utility

William I. Lundgren
 Gedae, Inc.
 1247 N. Church Street, Suite 5
 Moorestown, NJ 08057
 (856) 231-4458 x104
willundgren@gedae.com

James W. Steed
 Gedae, Inc.
 1247 N. Church Street, Suite 5
 Moorestown, NJ 08057
 (856) 231-4458 x102
jsteed@gedae.com

Kerry B. Barnes
 Gedae, Inc.
 1247 N. Church Street, Suite 5
 Moorestown, NJ 08057
 (856) 231-4458 x103
kbarnes@gedae.com

Abstract

The fundamental objective of MDD tools is to reduce monetary cost and schedule when developing and maintaining digital systems. We are interested in exploring the class of tools that automatically implement a digital application directly from a functional model. In these terms, the ultimate separation between functionality and implementation will recognize distinctly, a functional specification, a set of performance requirements, and a description of the target hardware with a tool that automatically constructs the implementation. In this ideal world, the functional specification would be fully verified independently of the target and would not have to be re-verified after the implementation. The table below

lists those parts of a digital application that lie in the functional and implementation domains. Real world applications are complex to the extent that automatically building such applications means that practical MDD tools must exhibit some marks of compromise. This paper explores measures that help assess where the value line lies between functionality and implementation in an MDD tool.

Coming back to the fundamental objectives, we need to define measures that focus our attention on the cost and schedule sensitive items. We define 4 measures useful for assessing the effectiveness of an MDD tool:

- 1) **Functionality:** Does a component or information affect the behavior of the digital system? If it does, it is part of the functionality. If it doesn't, it

Table 1 - Characteristics of Model Functionality and Realtime Implementation

Functionality: Signal and Data Processing, And Software State Control	Implementation: Realtime Performance
<p>Specification of the way the incoming sensor signal is to be processed to produce the desired results in the external environment:</p> <ul style="list-style-type: none"> • Process chain with conditional branches • Adaptive processing (feedback) • Store and retrieve data (database) • Maintain data state information <p>Mode control → control state of functionality in reaction to external state:</p> <ul style="list-style-type: none"> • State machine behavior • Switch functionality • Reset software state • Synchronize parameter changes with processing 	<p>Completion of functional processing to meet throughput and latency requirements:</p> <ul style="list-style-type: none"> • Distribution <ul style="list-style-type: none"> ○ Pipeline, Parallel, Load balanced, Round robin • Blocking / deadlock avoidance <ul style="list-style-type: none"> ○ Order, Queue size • Interprocessor communications <ul style="list-style-type: none"> ○ DMA, Socket, Shared memory, Broadcast • Execution granularity <ul style="list-style-type: none"> ○ Thread scheduling overhead ○ Memory requirements • Memory <ul style="list-style-type: none"> ○ Sharing, Fast vs. bulk • Compute density <ul style="list-style-type: none"> ○ Fixed chip architecture ○ Flexible (FPGA) architecture • Re-organize data in memory • Granularity of functional components

is part of the implementation.

- 2) Automation: without complete separation of functionality and implementation, there are compromises that vary in value impact. The questions that we pose are:
 - How much information must be supplied by the developer to specify the implementation?
 - Is the implementation information supplied by the developer stored separately so that multiple implementations can be realized from one functional model?
 - Does a change in the implementation force a re-verification of the functional model?
 - Is the automation complete? That is, are any changes required after the automation has finished?
 - Does the automation directly support an efficiency level sensibly comparable to a hand-coded alternative?
 - Is the automation aware of the parts of an implementation listed in Table 1?
- 3) Model Portability: Any changes necessary when a digital application is ported from one platform to another is an indication of limited portability of a tool. The platforms may differ in many dimensions. For example, the number of processors and processor capability might change. Anything that changes in the software is a part of the implementation not the functionality. There are 3 primary sources of non-portability:
 - Board architecture: How much does the functional model have to change if the functional model to processor mapping or interprocessor communication changes?
 - System architecture: How much does the functional model have to change if the external system interfaces (ADC, digital IO boards, etc.) change?
 - Chip (or CPU) architecture: Does the functional model have to change if the chip architecture changes?
- 4) Tool Portability: Ideally a tool should produce applications for every target hardware system without change. Practically the tool has some target dependent components used to build the implementation that must be ported to the new target. It may also have target dependent algorithms. We can measure the portability of an MDD tool by looking at 2 things:
 - How much effort is required to port the target dependent components?

- Does a completed port guarantee that every functional model will run on the new hardware target?
- Does the tool require modification when a new platform is targeted?
- Can the tool support every architecture?

In this paper we explore each of the measures of an MDD tool and the various compromises that are encountered. We focus particularly on the changes originated by the developer, the effect on the functional correctness and we discuss the associated costs in terms of monetary cost and schedule. Gedae is the MDD tool that gives us the basis for dissecting these issues.

Introduction

A key feature of MDD tools is the separation of functionality and implementation. A strong separation allows a majority of the implementation process to be fully automated, as well as model and tool portability. The implementation process of a MDD tool is shown in Figure 1. The functional and implementation specifications are stored in separate files and merged by the tool's compilation utility. This process creates an implementation that runs on a virtual machine. The amount of work required to specify the implementation is minimal and many implementations can be experimented with iteratively in a spiral development process. Thus, once the functionality is developed, it is easy to create a target-specific implementation and refine that implementation through experimentation.

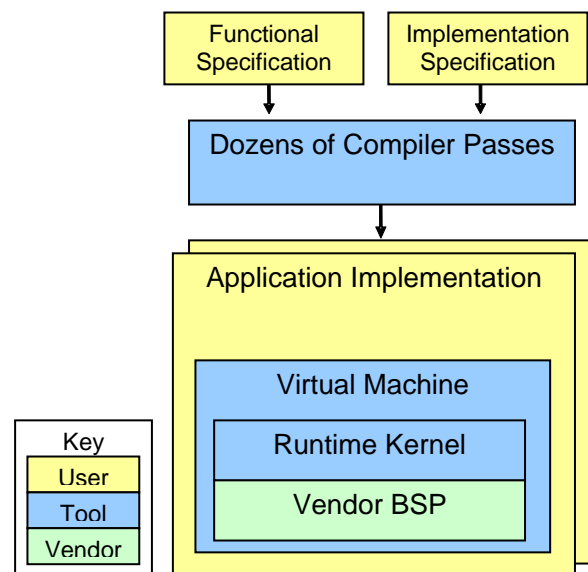


Figure 1 – Typical implementation process for MDD tools

This paper develops nine metrics for measuring strength. Seven metrics stem directly from the separation between functionality and implementation. The other metrics focus directly on the strength of the two components individually, one measuring the completeness of the language in being able to specify many applications, and one measuring the strength of the implementation in creating real world, real time applications. In addition to presenting these metrics, this paper will discuss key benefits associated with a strong score in each benefit.

Metric 1: Ease of Implementation

If the functionality has been developed, it should be easy to implement that functionality on the target system. In addition, it should be easy to tweak the implementation and rapidly test and analyze several implementations. Such a rapid testing of many implementations requires the implementation specification be strongly separated from the functionality specification.

As an example, consider processing granularity. Processors are often able to compute large vectors using homogenous operations more rapidly (in execution time per element) than small vector sizes. A typical method for improving an implementation to meet real time requirements is to rearrange the processing so that vectors are long by collecting multiple data sets in a buffer before initializing execution of the operation. This processing rearrangement is strictly part of the implementation. The processing granularity can be presented in a small, manageable set of parameters, and the granularity of specific threads can be easily increased to improve performance, as the application developer

analyzes the performance after each implementation is created.

MDD tools guarantee that the functionality will remain the same regardless of implementation. Thus, experiments like these can be performed quickly and easily, and developers should be able to iteratively develop an implementation that provides optimum performance.

Metric 2: Portability Between Processor Types

If the implementation is separate from the functionality, then the functionality should be able to be mapped to different processor types. If we study the implementation process presented in Figure 1, to change target processors, all we should have to change is the Board Support Package (BSP) portions of the virtual machine. If the BSP is changed for the new processor type, then the developer should only have to create a new implementation specification in order to map his functionality to a new processor type. Such a situation is shown in Figure 2; one functional specification is mapped to several unique DSPs.

The information contained in the BSP includes communication protocols, source code build process, libraries of optimized routines, and other items which define not just how to create an application on that processor but how to create an efficient one. Different vendors of the same processor type often create unique libraries and utilities to program those processors, therefore portability between the same processor type from different vendors must also be provided.

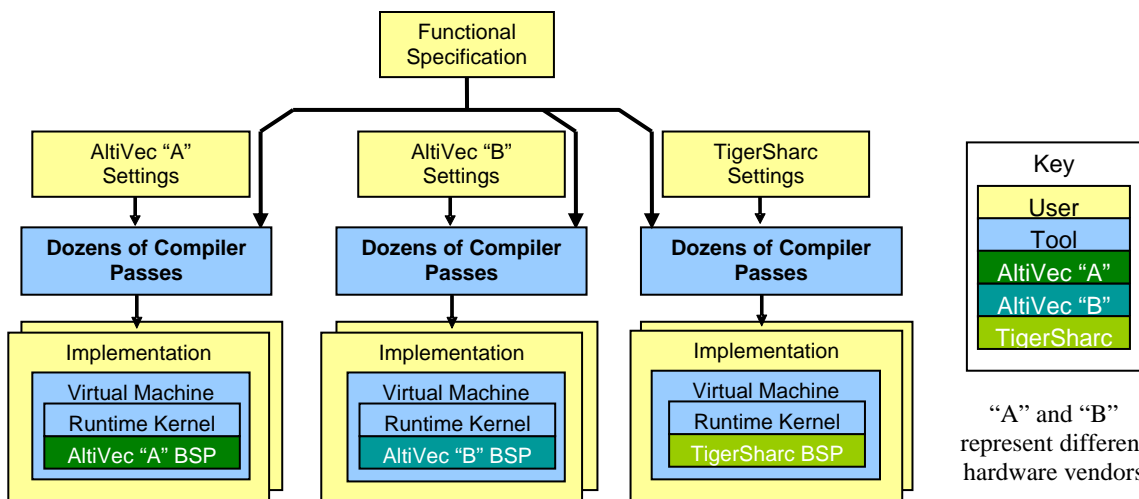


Figure 2 – The compilation and virtual machine allow for an implementation process which can be used on a wide variety of processors.

The BSP layer must be simple, generic, and independent of the application. It must contain much of the specialized knowledge required to program the processor (freeing the developer of these tedious details), but implementation of the BSP layer for a new architecture must also be a manageable task, and the BSP must be applicable to all projects, regardless of functionality.

If the developer can easily move his application to any processor type, then this provides much flexibility to the development team. First and foremost, much of the development can be done on a common workstation as the developer is assured that the functionality will be the same on his final target system. Furthermore, MDD tools can contain simulation environments for approximating the timing of the application on the actual target system before the implementation for that target is created. Also, this capability allows development teams to easily transition from COTS hardware to custom hardware. Custom hardware is often not available to the software development team until well into the development process. If the team can easily transition an implementation on COTS hardware to one on the custom hardware, then that allows them to more fully test and analyze their implementation before the final custom hardware is even available.

Metric 3: Portability to Multiple Processors

An extension to portability between processor types as discussed in the previous metric is portability to multiple processors. Systems sometimes cannot reach real time requirements running on a single processor. Some MDD tools can create implementations for multiple processors and handle all the interaction between processors automatically. When the tools do not support multiprocessor implementations, the development schedule is much longer. The development team must plan their application, design and simulate it on the workstation to validate the plan, then reimplement the system using multiprocessor code. This reimplementation may be a complete recoding, or it may involve integrating generated code from several toolsets and inserting communication into the generated code. If the MDD tool can handle the multiprocessor aspect, no reimplementation is required; the team must simply set parameters to partition and map the application and let the tool automatically create the multiprocessor implementation. Also, the MDD tool can provide a thorough analysis of deadlock avoidance, load, queue starvation, and other multiprocessor issues that can be difficult to address when integrating code from several toolsets.

Metric 4: Breadth of Functionality

Any application that can be created using hand-coding should be able to be created in the MDD tool. Many tools are specialized to only one type of processing, such as state machine control of application components or purely static signal processing of an infinite data stream, or to only one technology, such as only RADAR applications or only TELECOM applications. Ideally, the tool's programming language should be flexible enough that there is no limitation on the types of applications that can be developed. Many projects utilize a variety of tools – one for algorithm development, one for control using state machines, one for firmware development, etc. – and then go through a separate integration stage where output from the various tools is combined into the final implementation. Obviously, such an integration stage can be lengthy and full of risk, and it makes maintenance difficult as any improvements on one portion of the application must be reintegrated into the whole.

Metric 5: Strength of Implementation

The implementation that the MDD tool generates should be efficient. The tool should be able to create real time implementations. Ideally, the implementation's quality should be the same as hand-coded implementations (or higher), both computationally and in memory (and other resource) usage. Many system-level optimizations, such as memory reuse and thread execution planning, are difficult to achieve using hand-coding but are available to programming tools. The MDD tools should take advantage of these opportunities for optimization.

Additionally, the work of creating a strong implementation in the MDD tool should be small, e.g., setting a manageable set of parameters. If optimization requires special coding constructs and reorganization then the MDD tool may not be providing any benefit over hand-coding as it still requires detailed, specialized knowledge in order to create a usable implementation.

Being freed of tedious details, developers can focus on significant implementation issues, and users of MDD tools generally achieve significant gains in productivity. If an MDD tool can automatically generate production quality implementations with little effort from the developer, then these gains in productivity are not at the expense of application performance. This application performance does not

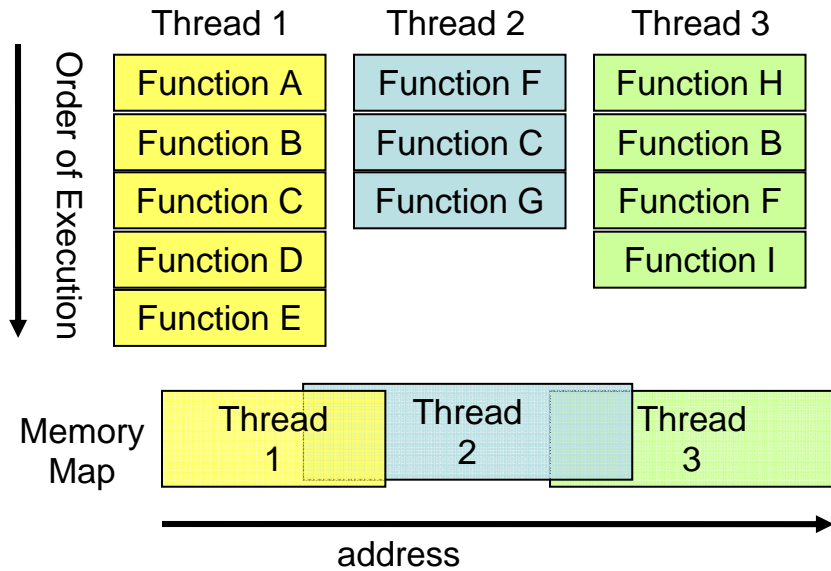


Figure 3 – If the MDD tool provides good visibility into the implementation, then details such as how the application is broken into threads and how the threads utilize memory can be analyzed.

One item is the structure of the threads, both how the processing is broken into threads and the order of execution of the functions inside each thread. Also, the memory usage and how the data for each thread is stored in memory, including any memory reuse, are useful for analyzing the implementation. If information such as these displays can be provided to the developer, then debugging can begin before the application is run on the target hardware and, with less effort, the developer is much more likely to be able to find places where the implementation can be optimized.

Metric 7: Visibility of Execution

just include real time requirements but also hardware costs, weight, size, and power. If the MDD tool is capable enough to tackle optimizations at the system-level then there is potential for the project to require fewer processors and less memory to meet its requirements.

Metric 6: Visibility of Implementation

The MDD tool creates an implementation based on the developer’s settings. In order to iteratively improve the implementation, the developer must be able to analyze what the tool has done and use this analysis to tweak his settings to improve the tool’s implementation. Figure 3 shows two such pieces of information the developer may need to analyze to gauge the quality of the generated implementation.

In addition to viewing how the tool has generated the implementation, it can also be very useful during debugging and optimization to view the performance of the system. Figure 4 shows such a display of timing events of system components. Timing information can be obtained through adding code to the functional model, but MDD tools can provide an organized, thorough display of all timing data, from processor load down to the timing of each primitive component. As shown in the figure, this information can make it easy to identify slow components that need to be optimized, places to alter the partitioning to improve the balancing of the load, etc. Also, this timing display can be used to show when the software is in different processing modes, problems in the interaction between processors (such as

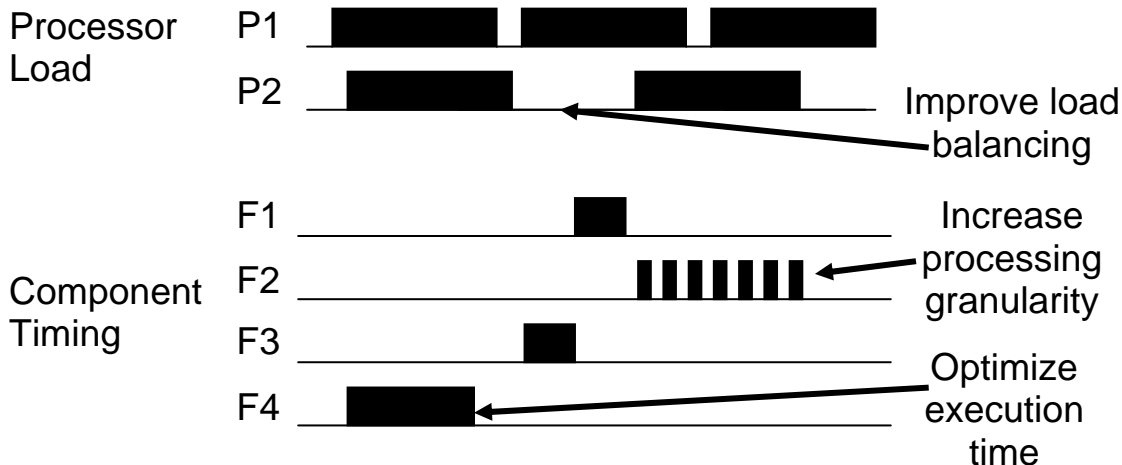


Figure 4 – A strong score in metric 7 allows for areas for optimization to be easily identified.

blocking on sends and receives), and other multiprocessor aspects to the execution which can be hard to visualize. Being able to view this information can enable an iterative development process, allowing an initial implementation to be quickly evaluated and easily identify what items need to be changed for the next iteration on the implementation.

Metric 8: Ease of Deployment

If a MDD tool is to support a wide variety of DSPs and embedded processors, as well as multiple such processors, then the tool should also automate the building and deployment of the application. Ideally one button press should

1. Generate C Code
Compile-able code for each component
2. Generate launch package
Code for managing threads
3. Generate makefiles
Scripts for building executables
4. Build executables for each processor
Run the generated build scripts
5. Load executables for each processor
Move the executables to each processor
6. Run all executables
Full deployed application is running

If the entire build and deploy process is automated, then the work of building, loading, and running executables is significantly reduced. Also, less specialized knowledge, such as how to compile and link an application for a DSP, is needed to run an application on the target processor.

Metric 9: Ease of Supporting New Target Hardware

If an MDD tool only supports a small number of hardware systems and cannot easily add support for new hardware or hardware from different vendors, then the gains from metric 2 (portability between processor types) are almost entirely nullified. If the functionality is to be mapped to a future hardware system, then support for the new hardware should be easy to incorporate. If this support cannot be added, then the functionality is not truly separated from the implementation. If this support takes a large amount of effort, then any productivity gains in moving to a target implementation is lessened by the extra work in producing support for the new hardware.

In the implementation process shown in Figure 1, the implementation process is the same for any target

because the vendor-specific and target-specific information is contained entirely in the BSP. In such a scenario, support for new hardware is reduced to a set, small number of steps in implementing the BSP for the new target. If a MDD tool performs strongly on this metric, then applications developed in the tool can be easily transitioned to new, state of the art hardware as it becomes available in the future with little extra effort. The implementation will still be efficient as optimizations applied to one target are applied to all targets and the BSP provides for enhancements like the use of optimized vector libraries.

Conclusions

This nine metrics present a strong basis for evaluated the utility of a MDD tool, or in fact any tool geared towards creating real world applications on deployed hardware. When approaching a new tool, the authors recommend evaluating it on a numeric scale on each of these metrics. Seven of the nine metrics stem directly from the separation between functionality and implementation which is the foundation of the MDD methodology. Metric 4 (Breadth of Functionality) and Metric 5 (Strength of Implementation) are the only two metrics that do not stem directly from this separation as they measure directly the capability of the tool at specifying a wide range of functionality or creating quality implementations. The companion paper to this discussion evaluates these metrics using Gedae, 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). The companion paper presents a real world search and tracking application and studies the application development process to evaluate how the tool performs on each metric.

References

1. W. I. Lundgren, K. B. Barnes, J. W. Steed, "Gedae: Auto Coding to a Virtual Machine", *High Performance Embedded Computing Conference*, 2004.
2. W. I. Lundgren, K. B. Barnes, J. W. Steed, "Implementing Mode Control on Multiple Processor Systems", *GSPx Conference*, 2005.
3. J. W. Steed, W. I. Lundgren, K. B. Barnes, "Autocoding Sensor Processing Applications to Run on Multicore Processors", *GSPx Conference*, 2005.