2014 NDIA GROUND VEHICLE SYSTEMS ENGINEERING AND TECHNOLOGY SYMPOSIUM
VEHICLE ELECTRONICS AND ARCHITECTURE (VEA) TECHNICAL SESSION
AUGUST 12-14, 2014 - NOVI, MICHIGAN

THE BENEFITS OF MODEL-BASED DESIGN AN LHP SOFTWARE TECHNICAL BRIEF

Dr. Steve Fraser
LHP Software
Columbus, IN

David Fenstermacher
LHP Software
Chicago, IL

Chris Doyle
LHP Software
Columbus, IN

ABSTRACT

Use of the Model-Based Design (MBD) processes is becoming increasingly common in embedded control system software as a means to manage software complexity, improve quality, and reduce development costs. The MBD process can achieve these goals by combining the design, simulation, and implementation of software features into a single, integrated workflow that reduces development effort and allows extensive software testing to be performed in simulation. In order to realize the full benefit of MBD, engineering organizations must invest resources intelligently in the tools, processes, and infrastructure to avoid common mistakes and pitfalls.

INTRODUCTION

Model-Based Design (MBD) is a software development
methodology in which the implementation, verification, and documentation of software features flow directly from a single description of the software behavior. The behavior of a software feature is designed, or "modeled," in a graphical environment, such as Mathworks' MATLAB/Simulink or National Instruments' LabVIEW, which allows the software design to be simulated in a virtual environment (Model-in-the-Loop Simulation).

The embedded, real-time implementation of the software design is generated automatically from the model, providing a hard link between the design, simulation, and implementation of a software feature. Finally, the graphical model provides a description of the algorithm that is easily understandable, without knowledge of a specific programming language. According to a study by Siemens [1],companies transition to MBD to (1) reduce software development time and resources, (2) gain capability to detect software errors earlier, (3) improve communication with colleagues, and (4) reduce effort for reusing software features in other deployments. The MBD process offers the potential to provide enormous benefits in all of these areas, but a successful deployment of the MBD paradigm requires that an organization invest in the correct MBD tools, processes, and infrastructure. A successful MBD development environment must be designed to encourage consistent and readable models, hardware-independent software models, and reusability of software, all while maintaining a straightforward development process from the perspective of the software design engineer.

BEST PRACTICES FOR MODEL DESIGN

A. The Importance of Proper Model Design.

In the MBD process, the software model becomes the sole description of a given software feature; therefore, it is extremely important to maintain the quality and integrity of these models. Well-designed models produce more efficient software and more meaningful simulation results with less effort. Additionally, a well-organized model can provide extremely concise and up-to-date documentation of the algorithm that is understandable to all stakeholders in the software development process, including non-programmers. Done well, the MBD process can greatly reduce the need for written software documentation.

B. Managing Signal Flows

The core benefit of graphical software modeling tools is the ability to show data flow between logical elements in an intuitive, graphical form. Ironically, however, presenting data flows in an easily understandable fashion is also among the greatest challenges in an MBD environment. While signal flows can be presented as lines in simple models, modern embedded software applications must manage large numbers of signals, often making it impractical to represent them all with individual lines. Signal lines can be broken or combined to save space, but this also breaks the data flow representation and reduces the intuitive nature of the graphical representation. The best model design practice is to encapsulate adjacent model elements into subsystems, such that each subsystem has an obvious function but the smallest possible number of inputs and outputs.

C. Model Design Standards

Many of the most important model design practices, such as maintaining logical signal and execution flows, are quite obvious to any software designer. However, these best practices can often be forgotten by software designers when they are under pressure from deadlines or struggling to make a complex feature work correctly. Poor adherence to good model design practices may allow for faster initial model development, but, in the long term, will lead to models with extremely confusing signal and execution flow that are unmaintainable and too interconnected to simulate effectively. To avoid this problem, every software organization should develop and enforce a clear standard for the appearance and organization of software models. The standard should detail naming conventions, which blocks or constructs are acceptable or unacceptable, best practice designs for common tasks, etc. Model design standards not only improve the quality and long-term maintainability of models, they also improve communication across an organization through a consistent model format.

D. Library Elements

The use of library elements also improves the ability to maintain model consistency and quality across an organization. When the same logical construct is required over and over throughout many models, the logic can be designed once and placed into a library element which can be reused anywhere in the software design. Library elements save time when designing a feature, and improve quality by providing a single “correct” way to implement common constructs.

SOFTWARE ARCHITECTURE AND TEAM ORGANIZATION

A. Component-Based Software Architecture

Many embedded software projects contain a large (and constantly growing) number of features. Because of the complexity of the requirements and the resulting large amount of software logic needed to implement them, it is often impractical for the software design for an entire system to be placed in a single model file. Formal partitioning

of software models into smaller "software components" provides a structure to maintain coherency of data flow throughout the software and enables many engineers to work in parallel on different software features. All data moving into and out of the component must pass through a pre-defined, documented interface, thereby enforcing a certain amount of organization in the software base. This encapsulation prevents software components from directly accessing data from other components or operating system (OS) software, which maintains hardware-independence and allows for much easier reuse of the code across platforms and applications.

B. The Development Environment

While MBD has the potential to dramatically increase productivity by abstracting away the details of the software implementation, it invariably reduces the flexibility of the designer in implementing novel constructs, and there is a potential for the applications engineer to become bogged down in overly complicated models. To manage this problem, an MBD system needs to migrate to a different resource paradigm. In a traditional development environment, the systems engineer designs the expected behaviors of the system and the software engineer writes embedded code based on the design, often with little knowledge of the underlying principles behind the design. In an MBD development environment this is reversed: The application engineer simultaneously designs and implements the software logic, whereas the software engineer creates an infrastructure that allows model-generated code to integrate and interact with other software features and with the target hardware platform. This infrastructure layer abstracts the hardware functionality, such as hardware I/O and datalink communications, into a format that is usable in software models, and provides reusable utilities, and palettes for common operations. These infrastructure elements may be implemented as external hand coded C/C++, by higher order models of native elements, by integrating external scripting languages, or even external object libraries. Rather than coding specific application designs, software engineering resources are reallocated to producing optimized low level implementations that are reusable across a variety of applications. For example, from the perspective of the applications engineer, the inclusion of a specific J1939 message can be as simple as pulling a PGN element from a library and selecting the data element to pull out from a port. The low level implementation of the code to accomplish this task is developed beforehand using the most efficient implementation (be it MBD primitives or C code), implementing only the sufficient functionality that is needed by the application requirements.

C. Functional Reuse of Model-Based Designs

In the MBD development environment, the reusability of software elements is enhanced in a number of ways. By abstracting the application design, the deployment does not have to be optimized to a specific hardware platform. Only at code generation is the design optimized for a specific platform, which will include tradeoffs between maximizing throughput versus RAM and flash space, use of floating point verses fixed point data implementations, and taking advantage of compiler optimizations. The graphical nature of the code can also make it easier to integrate software components and modules. However, according to a recent study [2], MBD alone does not improve model reuse unless a reuse concept is applied. When a model becomes too big or is poorly structured, it becomes difficult to cut and paste without having to re-work significant portions of the original implementation. If componentized software architecture is used, as described in Section A, many software components can be reused without modification.

D. Not All Implementations are Suited to MBD

The use of MBD as an algorithmic design tool is most powerful when the application naturally follows a signal flow model. Not all software logic can be intuitively described in a signal flow form. MBD does not lend itself well to low-level hardware interface and device driver implementations, nor generic data parsing such as the implementation of communications protocols and CLIs. In addition, heavily iterative processes and dynamic resource allocations do not fit well with the physically representative nature of graphical models. An effective MBD embedded process will use a combination of graphical and text-based design tools as appropriate. Elements that themselves cannot be implemented as models are typically linked to library elements that can be referenced in the MBD environment.

MODELING, SIMULATION, AND CODE GENERATION TOOLS

The primary tool required for an MBD environment is the algorithm design tool, which provides a graphical interface to construct software models. Since MATLAB/Simulink is the de-facto standard for development of real-time embedded control algorithms in most industries, many of the third party tools for code generation, simulation, and testing are either based on it or compatible with it. National Instruments' LabVIEW is also used to develop simulations and software for some embedded applications. LabVIEW does not have as many third-party compatible tools, yet, National Instruments provides a fairly comprehensive suite of code generation, simulation, and verification tools.

MODEL INTEGRATION

In most cases, multiple software behavior descriptions (either models, hand-written C code, or object code) from multiple sources must be combined in order to simulate the entire system, or to build the complete software for the target hardware platform. Due to the complexity of many embedded software projects, it is usually desirable to have a software tool to automate the integration process. Such an integration tool should allow the user to define interfaces to application software components and low-level OS functions and utilities, and to abstract the process of integrating models and code into larger simulations and software builds. UML-based tools provide an application-generic method for high-level software design and simulation in a C/C++ environment, but the component wrappers that they create are not well integrated with the graphical tool environment. Specialized commercial software, such as dSPACE SystemDesk or the Vector DaVinci suite for AUTOSAR systems, suffers from the requirement that they be generic and agnostic to tool selections. Very often the optimal solution is a custom designed system integration tool that provides build integration and simulation capabilities as well as configuration management. Control and data flow can be made visible by giving the integration tool the capability to generate UML views of the system.

SIMULATION AND VERIFICATION

One of the most powerful elements of MBD is the ability to test software at nearly all levels of the software development process. However, adapting test procedures to an MBD approach requires a top down reconsideration of the entire software test process, including unit/component testing, integration testing and system testing, and exploiting the power of simulation at every level. Since the models themselves can be simulated, software designers can test the software directly on their PC during the design process. Similarly, simulation testing can be performed at all levels of scope, from a small unit test of a single feature all the way up to a simulation of the entire software system against a sophisticated plant model. This testing can even be automated to a

great extent, thus eliminating much of the tedious work involved in software verification.

Studies have shown that most companies that migrate to an MBD software development process experience a front-end loading of test activities compared to the traditional design approach [1]. In order to achieve the benefits of MBD-based simulation testing in the long term, companies must invest resources up front to develop robust simulation testing tools, procedures, and plant models. After the initial investment, the resulting enhanced testing capability allows for more issues to be discovered early in the development process, before the software is ever run on the target hardware platform. Simulation testing not only reduces costs associated with testing software on-hardware, but also reduces overall software development time and costs. Large amounts of data can be recorded from a simulation to relay back to the software designer in the case of a test failure; this allows the designer to quickly locate and fix the issue. Additionally, the updated software can be immediately deployed back to the simulation environment. Surprisingly, the opportunity to find and correct problems in simulation testing is often missed or not leveraged effectively by many organizations because of reluctance to make the upfront investment in test infrastructure or because of poor software design practices. Testing must be considered throughout the software design process: The application software must be well abstracted from any low-level OS functions to allow hardware platform-independence, and the software designer must refrain from using model constructs that cannot be simulated (such as calls to hand-written software functions).

THE LHP COMPETITIVE ADVANTAGE

LHP Software was a pioneer in the adoption of MBD in the automotive industry and continues to be heavily engaged with Model-Based Design and development for a variety of customers. By working with multiple development platforms and toolsets, our engineers have a much broader and deeper experience than would be possible from working with a single platform in a single industry. LHP has extensive experience at all levels of the development process, from system architecture and process migration issues, tools and platform support, and adaptation to a customer environment. LHP continues to grow its capabilities by investing in proprietary processes and models and by experimenting with new technologies.

CONCLUSIONS

Adoption of the Model-Based Design process allows enormous potential improvements to software quality, cost, and development time. The time required for software development is significantly reduced by condensing the design and implementation of most software into a single step. The enhanced simulation capability of the MBD system allows for software problems to be detected much earlier in the development process, before the software is ever run on the target hardware platform. Software models also provide an extremely concise and always up-to-date description of the software behavior that is accessible even to non-programmers, thus facilitating easier communication and understanding of software concepts throughout an organization. When the best practices for MBD software design and architecture are followed, the resulting software is packaged into hardware platform-independent software components that are easily re-used across applications.

Although the potential benefits of MBD are substantial, the process of creating an MBD infrastructure involves many difficult choices that must be carefully considered to avoid common pitfalls. Before making the substantial investment needed to implement MBD, an organization should develop a deep understanding of the MBD process and an appreciation for the sometimes

painful lessons learned. Choosing suppliers and development partners with significant experience in MBD is an excellent way to leverage the existing knowledge base within the industry to ensure a successful deployment of the MBD process.

REFERENCES

[1] Fieber, F., Regnat, N. , Rumpe, B., "Assessing usability of model driven development in industrial projects",Siemens AG, Corporate Technology CT SE 1, Otto-Hahn-Ring 6, 81739 München, Germany

[2] Kirstan, S., Zimmermann, J., "Evaluating costs and benefits of model-based development of embedded software systems in the car industry – Results of a qualitative Case Study", ALTRAN GmbH & Co. KG, Bernhard-Wiki-Str.3, 80636 Munich, Germany.