Software Development Principles Applied to Graphical Model Development

Paul A. Barnard*
The MathWorks, Natick, MA 01760, USA

The four fundamental principles of good software design—communicate clearly, think in modules, automate where possible, and test early and often—are as relevant today as they were thirty years ago. Until recently, however, they were confined to software development. Model-Based Design has broken down that barrier. Modeling and simulation have long been key components of aerospace vehicle development. The increasing complexity of aerospace systems is driving a need for high-fidelity simulations of the vehicle and the control algorithms during multiple program phases. By using graphical modeling techniques and Model-Based Design, design engineers can create dynamic system models for simulating the vehicle and the control algorithms and use those models at every stage of development—from requirements capture to design, implementation, and test. Companies that have adopted Model-Based Design report time and productivity increases as high as 60 percent. Model-Based Design brings dramatic results not only because it streamlines the workflow, but also because it brings the principles of good software design into the world of the control and signal processing engineer.

I. The Evolution of Software Design Principles

Software engineering has had a long history in the aerospace and defense industry. Since the days when a mainframe computer filled a room and simulation results were delivered as ten-inch-thick printouts, the industry has recognized the necessity to write good software; for example, many of the procedures invented in the 1960s to program the digital control system of the lunar module's Guidance and Navigation system[1], such as state-space modeling and optimal control techniques, are now a standard part of today's software engineering process. Software developers have continued to hone the art with planning methods, tracking metrics, and development tools[2], to the point where aerospace uses some of the more disciplined processes running at the highest quality levels.

In the 1960s and 1970s, the standard was to program in assembly code. As embedded hardware processors became more powerful, the software that runs on them expanded to provide increasingly complex functionality. To deal with this complexity, the software community adopted a higher level of abstraction, moving software writing from the "machine" level to the "programming" level through languages like C and FORTRAN. Software engineers learned this higher level of abstraction and then used automation (a compiler) to automatically create executable software. Now we are seeing a move to the next level of abstraction (system-level design) and automation (code generation), increasing the scale of systems that software engineering can tackle.

As the demand for high-integrity, safety-critical software intensifies, software engineering methodologies and tools continue to improve efficiency and quality and enable developers to react more quickly to changing requirements. The most recent methodologies, Extreme Programming and Agile Development, focus on improving communication within organizations and with customers through working software rather than multitudes of documents; writing tests very early in the process, even before any code is written; and starting projects with simple designs that evolve to incorporate full capability[3,4].

A review of the history reveals that four key principles have driven many improvements in software engineering:

- Communicate clearly
- Think in modules

* Marketing Director, Control Design, 3 Apple Hill Drive, AIAA Member

- Automate wherever possible
- Test early and often

Until recently, these principles were confined to software engineering departments. The control and signal processing engineers often failed to understand, appreciate, or embrace these techniques, causing inefficiencies and friction between the groups. For example, the control and signal processing engineers would talk of "throwing specs over the wall," a way of saying that they did not know what was happening in the software world. This lack of awareness of the principles that drive good software design is easy to understand: frequent testing between algorithm and software development is impractical when physical prototypes are prohibitively expensive or unavailable; ambiguous paper specifications impede communication; and automation is limited when there is a lack of tools to support it.

The latest graphical modeling tools and techniques have fundamentally changed embedded system design methodology by putting modeling and simulation at the center of system design. The result, Model-Based Design, brings the sound principles of software development into the world of the control and signal processing engineer.

II. Working with Model-Based Design

Model-Based Design is not a process per se, but has been applied in many different design flows, from the "V" diagram to the waterfall and the spiral. It can begin as soon as project requirements are available. These are captured in a block diagram model that encapsulates all system dynamic characteristics, including environmental components such as actuators, sensors, mechanical devices, electronics, and other physical elements.

During requirements capture the model is an idealized representation of the system. Details are added as the design progresses until the model becomes an "executable specification" that includes all the information needed to specify the software or hardware implementation, including fixed-point and timing behavior. Because the model is also the documentation, designs can pass directly from the desktop to pilot training simulations without introducing errors and with minimal manual effort. And because the model is an executable specification, engineers can automatically generate code from the model for real-time prototyping and deployment in the target system. Like the model, the code can be tested and verified at any point. Problems are easily corrected by adjusting the model and regenerating the code, maintaining specification integrity between the model and the code.

The new generation of graphical tools—for example, the Simulink® product family from The MathWorks—provide unified modeling environments that include interactive graphical editors, graphical debuggers, and tools for model analysis and diagnostics[5]. Their customizable block libraries enable engineers to accurately design, simulate, implement, and test control, signal processing, communications, and other time-varying systems. Complex designs are more easily managed because the designer can segment models into hierarchies of design components and break them down into functional elements of arbitrary size and structure. Designers can achieve multiple levels of model fidelity simply by substituting one model element for another. This approach streamlines large-scale system development by letting multiple design teams work in parallel to refine and optimize subsystem design.

Organizations that have adopted Model-Based Design report spectacular results: the flawless functioning of thousands of lines of automatically generated flight-control code, productivity improvements of 500%, development time cut in half[6]. Results like these are possible not only because Model-Based Design offers a development paradigm that improves development speed, communication, and efficiency, but also because it fosters good design practices at every stage, from requirements capture and design to implementation and test.

III. Applying Software Design Principles

The four key principles of software engineering discussed above are evident within Model-Based Design. For those using Model-Based Design, these principles are naturally encouraged through the development tools within an organization. Here we review each of the principles and look at how Model-Based Design fosters them in aerospace applications.

A. Communicate Clearly

Good communication within and across teams is critical when requirements are evolving and technology is being developed for the first time, as is typical in aerospace applications. In a traditional design process, documents are the primary means to communicate requirements, specifications, test scenarios, data definitions, and other project details. Document-based approaches are subject to ambiguity and misinterpretation. They encourage a flow that is unidirectional, preventing rapid iteration of a design and reinforcing the wall between software development and algorithm design.

Software methodologies such as Extreme Programming address this problem in two ways: by having programmers work closely together within teams and by making the primary communication vehicle of a team's output working code, not documents[3].

Model-Based Design applies the same principles to control design and signal processing because it is predicated on an executable, graphical model being the repository of design information. Using graphical diagrams instead of textual code for the design and implementation improves communication within and among design teams.

Most development processes can be broken down into four phases, as shown in Figure 1: Requirements and Specification, Design, Implementation, and Test and Verification. The model is a key communication vehicle in each phase. Starting as a closed-loop system that captures requirements, it receives design input through multiple design iterations, is further refined with detailed software engineering constructs appropriate to the target processor; and incorporates static and dynamic test vectors and acceptance criteria. In short, the model is the specification, the design, the implementation, and the test cases.

In addition to streamlining communication between project phases, graphical diagrams are easier to scan and read than text documents because they are much less ambiguous. A system requirement can easily be omitted from a document by accident. When system behavior is being captured in an executable model, however, the designer is forced to provide enough detail for the system to be simulated, ensuring that major requirements are included. Even if a system is simply a "stub" node, the designer will have made a conscious effort to build that stub and will not simply have forgotten about it.

For example, consider an aircraft automatic landing system that uses a radar altimeter as one of its components. The guidance algorithms and the radar altimeter are being developed by separate groups working in parallel. The group that is developing the guidance algorithms to keep the aircraft on the desired glideslope to the runway doesn't yet know the detailed dynamics and characteristics of the radar altimeter. For this example, let's assume there is concern over the reliability of the radar altimeter signal.

A high-level model of the system is built so that the guidance group can proceed. They build a "stub" model of the radar altimeter called "Radar Altimeter". Because of the concern over the quality of the data from this system, they add to this a "signal quality" output and build this signal into their processing algorithm, making sure that it has the proper data update rate and so forth. The Radar Altimeter block can now be the interface specification for the radar group to build to. There is no ambiguity in this specification. The signal dimensions, data types and data rate are all inherent in the signal definition, since a working simulation is involved. This is in contrast to paper specifications, which could easily omit important details. Aerospace systems typically involve this type of coordination between multiple engineering disciplines.

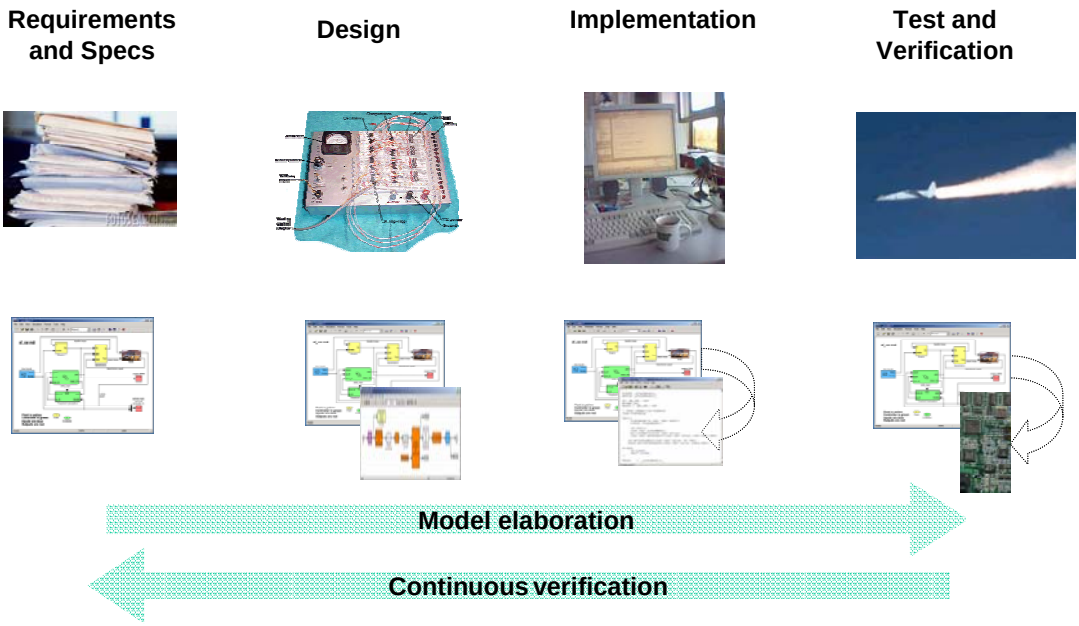

Figure 1. Model-Based Design enables communication between project phases via models.

B. Think in Modules

Software developers know that complexity can be reduced by building clear and simple modules that serve specific purposes. Newer software development techniques like Extreme Programming tell us that designs should be a simple as possible while supporting the required functionality[3], and standard software metrics such as cyclomatic complexity encourage reduction of the number of paths through a software module[2].

Model-Based Design fosters modularity because it begins with a working (simulatable) model and because the APIs between subsystems are rigorously defined. Graphical models are typically hierarchically organized to hide complexity on the computer screen. This naturally fosters a modular approach: The limitations of a typical computer screen encourage the designer to organize the blocks and systems on the screen into a hierarchy of subsystems. With a text-based programming language, on the other hand, the designer might continue to write code in a linear fashion, scrolling through page after page until it is difficult to see the important sections of the program.

A modular approach does not force the designer to work at a high level of abstraction. The level of detail included in the modules can be adapted to the phase of design work. In early phases, simple subsystems revealing only a shell of the actual algorithms will be built. In later phases, detailed algorithms incorporating target-specific information will be included. But even at the final stages of a project, the team should still be able to view the initial, simple design at the high levels of the hierarchical model.

Consider the model for a missile system with an electrically powered control fin actuator. Figure 2 shows the airframe dynamics, including the control actuator. This model is implemented in Simulink[5] and Stateflow®[7]. It shows a configurable subsystem being used to encapsulate the details of the actuator. Note that the currently selected actuator model is a second-order linear model. Other subsystems for the aerodynamic, equations of motion, and autopilot are also shown. Figure 3 shows details of two of the three options for actuator models.

A typical project might start with a simple linear actuator for initial design to work out the overall control scheme. As the project moves into design, more detailed non-linear actuator models might be employed. For final system sizing, a detailed electrical model of the motor would be built to determine if the actuator has the torque and bandwidth to meet the needs of the control system. Because all these models conform to the API that was built with the original, simple model, test cases built early in the process can be reused throughout the design.

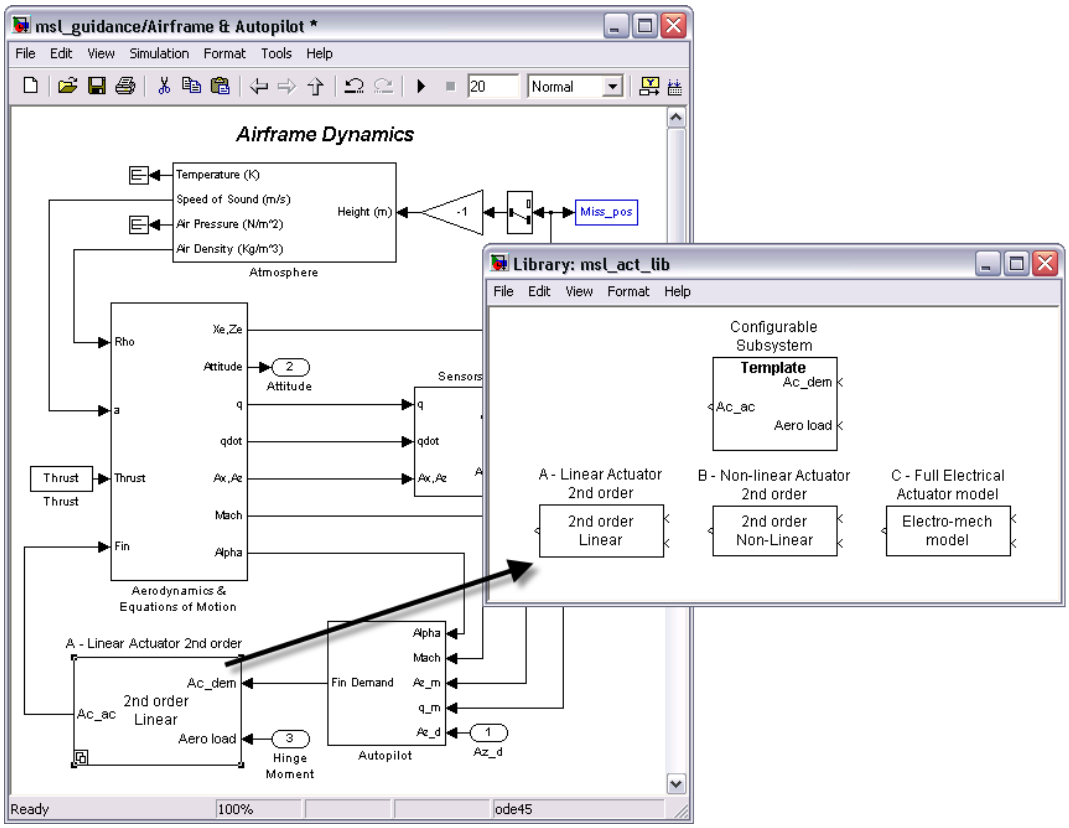

Figure 2. Modularity encapsulates detail and standard APIs to enable varying levels of fidelity. In this generic missile model, the arrow shows a single subsystem with a defined interface linked to three different actuator models used at different development stages.

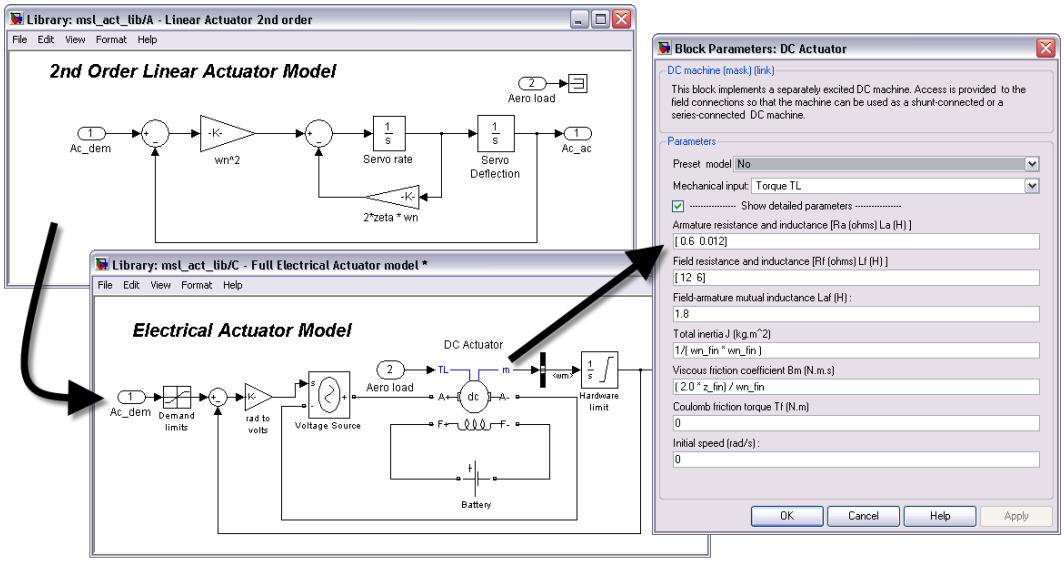

Figure 3. A simple and detailed missile flight control actuator model share the same standard interface as the rest of the model.

C. Automate wherever possible

Automation not only saves time; it applies rules and processes consistently across a software application much more effectively than a human team. The most dramatic improvements in software engineering have come through automation. Using a compiler and linker to automate the generation of machine code from a higher level language like C enabled software engineers to produce working systems in less time. Other automation areas, like generating documentation automatically, checking coding style, and developing unit tests, continue to evolve.

Model-Based Design embraces automation primarily through automatic code generation from models. Generated code can be used for a number of purposes, including simulation, real-time rapid prototyping, hardware-in-the-loop simulator implementations, and final flight code development. Software elements, such as discretized blocks, fixed-point data types, and links to existing "legacy" code, are added to the design model according to the purpose of the code. Model coverage and profiling tools then automatically analyze the model and generate code for test completeness and performance. This code is generally ANSI or ISO C code, but it may include tailored sections for interfacing with hardware device drivers or other software elements.

With Model-Based Design, the same models can be used to automatically generate design documentation as well as the code. Standard APIs to the models can used to build tools to automatically check that models conform to organizational style guidelines. In recent years, there have been significant efforts to develop standard guidelines for users of Model-Based Design[8].

An example of how automation improves coding efficiency is shown in Figures 4 and 5. This model demonstrates "expression folding" performed by Real-Time Workshop®[9]. In this model, each block (gain, lookup tables, relational operator, logic, and constant) is folded into the switch block operation. Expression folding dramatically improves the efficiency and readability of the generated code. Note that the branches of the switch block are conditionally executed, increasing CPU throughput.

This type of optimization is very valuable for an individual subsystem, but its effects are magnified when applied to a large system model of 10s or 100s of thousands of blocks. Applying this optimization across a large model can result in code that is more efficient than if it was hand-written by a large team. The reason this is true is that these types of relatively simple optimizations (expression folder in this example) can be applied consistently, through automation, to the entire project – something that is difficult to achieve through human interactions.

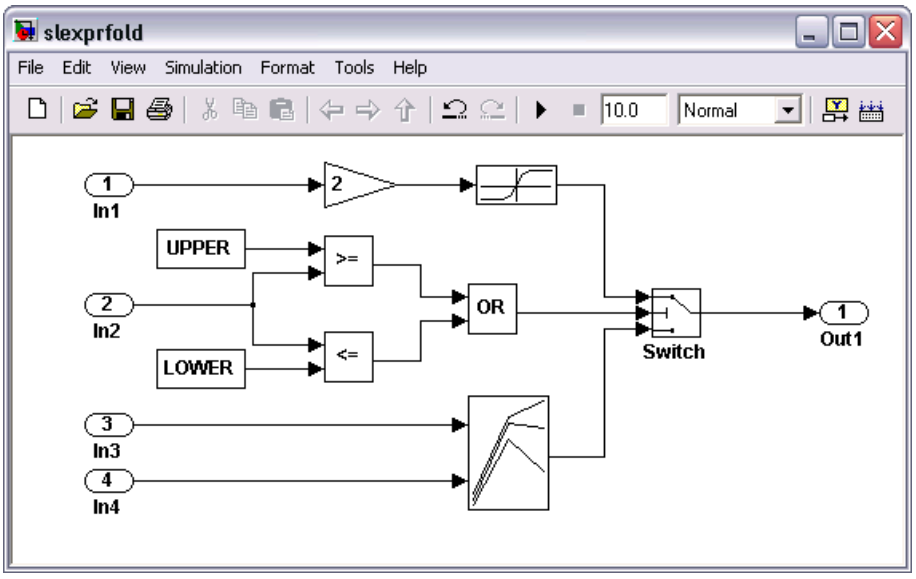

Figure 4. Typical switch logic subsystem with logical expressions and lookup tables.

```
/* Switch: '<Root>/Switch' incorporates:
 *  Gain: '<Root>/Gain'
 *  Lookup: '<Root>/Look-Up Table'
 *  Lookup2D: '<Root>/Look-Up Table (2-D)'
 *  RelationalOperator: '<Root>/Relational Operator1'
 *  RelationalOperator: '<Root>/Relational Operator'
 *  Logic: '<Root>/Logical Operator'
 *  Constant: '<Root>/Constant1'
 *  Constant: '<Root>/Constant'
 *  Inport: '<Root>/In4'
 *  Inport: '<Root>/In3'
 *  Inport: '<Root>/In2'
 *  Inport: '<Root>/In1'
 */
if((rtP.UPPER >= rtU.In2) || (rtU.In2 <= rtP.LOWER)) {
  rtb_Switch = rt_Lookup(rtP.T1Break, 11, rtU.In1 * 2.0, rtP.T1Data);
} else {
  rtb_Switch = rt_Lookup2D_Normal(rtP.T2Break, 3, rtP.T2Break, 3,
                    rtP.T2Data, rtU.In3, rtU.In4);
}
```

Figure 5. A portion of the code automatically generated from the model in Figure 4, with expression folding optimization applied.

D. Test Early and Often

Modern software development principles focus on testing early. In fact, Extreme Programming advocates that testing should be done "all the time[3]." Other modern techniques, such as Test-Driven Development, talk about building the tests before any code is written. In this process one would write tests, then write code, and then refine the code[10].

Test and verification of embedded systems is traditionally the final step before product delivery. Errors uncovered this late often require looping back to early stages in the process, resulting in project delays and cost overruns.

Model-Based Design moves validation into the early part of the software development cycle, reducing the risk of late error detection. Engineers can integrate tests into the models at every development stage and can quantify test coverage of the model. This continuous verification and simulation helps identify errors early, when they are easier and less expensive to fix, and streamlines final verification.

Model-Based Design encourages early testing behavior because a working (simulatable) model is built and available at the very beginning of design work. Each time a model is simulated, it is essentially tested. And, if the tests are built at the same time as, or even before, components, those tests verify the operation of the design as development proceeds. Figure 6 shows how test cases and design models are used together to enable continuous verification in a development process.

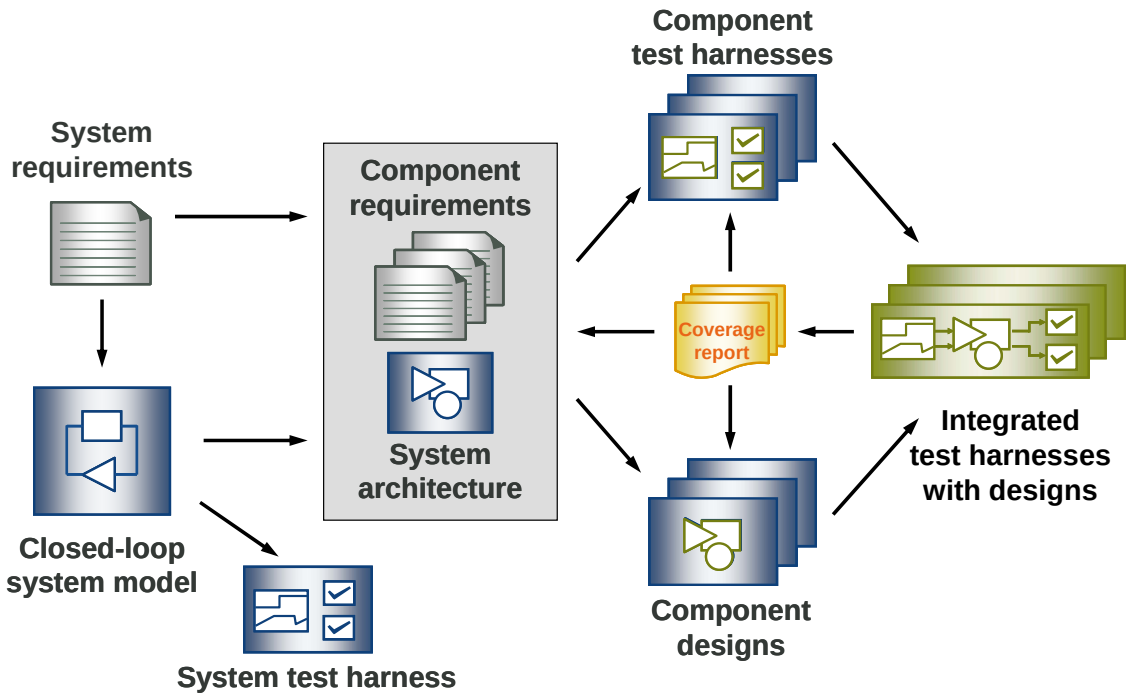

Figure 6. With Model-Based Design, tests can be built in conjunction with component design models, enabling verification throughout the development process.

IV. Conclusion

Model-Based Design (Figure 7) is enabling aerospace companies to develop high-integrity software-oriented systems through the use of executable graphical models and automatic code generation to:

- Define executable specifications
- Use multi-domain models to design embedded systems
- Generate code automatically
- Test and verify continuously

In this way, Model-Based Design fosters the four fundamental principles of good software design and puts them within easy reach of control and signal processing system engineers.

Design tools and technologies continue to evolve, enhancing the ability of Model-Based design to accelerate system engineering processes. Model-style and semantic-checking tools will enable increasingly effective and agile management of requirements, specifications, and designs. The multi-domain simulation aspects of design will continue to be important as systems add more integrated technologies, and test definition, execution and management will remain critical to the verification and validation of safety-critical algorithms.

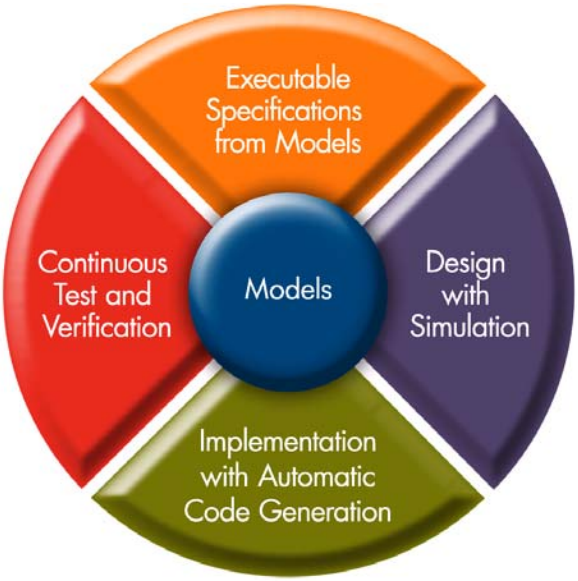

Figure 7. Model-Based Design enables four key technologies centered on executable, graphical models.

References

[1] Gran, Richard J. "Fly Me to the Moon: Then and Now." *MATLAB® News & Notes*, Summer 1999. http://www.mathworks.com/company/newsletters/news_notes/sum99/index/html

[2] Pressman, Roger. *Software Engineering: A Practitioner's Approach*. New York: McGraw-Hill, Inc., 1992.

[3] Beck, Kent. *Extreme Programming Explained.* Reading, Massachusetts: Addison Wesley Longman, Inc., 2000.

[4] Fowler, Martin. "The New Methodology." http://www.martinfowler.com/articles/newMethodology.html

[5] Simulink®, Software Package, Ver. 6.2.1, The MathWorks, Inc., Natick, MA, 2005.

[6] MathWorks User Stories. http://www.mathworks.com/applications/controldesign/userstories.html

[7] Stateflow®, Software Package, Ver. 6.2.1, The MathWorks, Inc., Natick, MA, 2005.

[8] Erkkinen, T., "Model Style Guidelines for Flight Code Generation," *AIAA Modeling and Simulation Technologies Conference*, AIAA-2005-6216, AIAA, San Francisco, CA, 2005.

[9] Real-Time Workshop®, Software Package, Ver. 6.2.1, The MathWorks, Inc., Natick, MA, 2005.

[10] Vautier, Eric, and David Vydra. "Test-Driven Development." http://www.testdriven.com/modules/news/