SAE TECHNICAL PAPER SERIES

2006-01-1357

Executable Specs: What Makes One, and How are They Used?

Peter J. Schubert
Packer Engineering, Inc.

Lev Vitkin and Frank Winters
Delphi Electronics & Safety

ISBN 0-7680-1633-9

9 780768 016338

SAE International™

2006 SAE World Congress
Detroit, Michigan
April 3-6, 2006

400 Commonwealth Drive, Warrendale, PA 15096-0001 U.S.A. Tel: (724) 776-4841 Fax: (724) 776-5760 Web: www.sae.org

All SAE papers, standards, and selected books are abstracted and indexed in the Global Mobility Database.

ISSN 0148-7191

2006-01-1357

Executable Specs: What Makes One, and How are They Used?

Peter J. Schubert
Packer Engineering, Inc.

Lev Vitkin and Frank Winters
Delphi Electronics & Safety

Keywords: Executable Specifications, Systems Design, Model-Based Development, Simulation and Modeling.

ABSTRACT

Model-based systems development relies upon the concept of an executable specification. A survey of published literature shows a wide range of definitions for executable specifications [1-10]. In this paper, we attempt to codify the essential starting elements for a complete executable specification-based design flow. A complete executable specification that includes a functional model as well as test cases, in addition to a traditional prose document, is needed to transfer requirements from a customer to a supplier, or from a systems engineer to electrical hardware and software engineers. In the complete form demonstrated here, sub-components of a functionally-decomposed system manifest as modular reuse blocks suitable for publication in functional libraries. The overarching definition provided by product architecture and by software architecture must also be harmoniously integrated with design and implementation. Using seven specific automotive examples, we illustrate effective ways in which executable specifications have been used in production-ready applications. Benefits of model-based development are captured, including earlier and more thorough testing, automatic document generation, and autocode generation.

PROBLEM DESCRIPTION

Systems engineering begins with requirements. In a traditional design approach, the systems engineer decomposes customer-level requirements into functional subsystems. The subsystems will derive their requirements from the top-down, augmented by internal standards for systems, software and hardware design, from government regulations and from constraints imposed by the system hardware itself. This process of functional decomposition continues to lower levels of abstraction (hierarchy) until each subsystem is atomic – where an individual or small team can implement it without further clarification.

This traditional design approach is effective, but not necessarily the most efficient means of product design. Inefficiencies arise from the inability to objectively verify the design at each step of the process, thus pushing the thorough verification of the design to the end. The inability to electronically transfer, or re-use, any design implementation from the previous steps in the design flow is also inefficient. Most often, each step in the design flow starts as a “clean sheet of paper” with written documentation as the only entity transferred between design steps.

To accelerate engineering productivity of the entire design process, there is a widespread and growing adoption of model-based development approaches that include graphical representations of math-based simulation methods. The use of model-based methods enables engineers to gain a more thorough understanding of the details of a design much earlier in the development process. The model-based approach also allows the verification process to occur concurrently with the design process, thus improving the quality of the design from the start. In this case, both the design details and the verification methods can be electronically transferred through the design flow, enabling better communications and data transfer throughout the design process.

One widely-used term used to describe model-based development is an executable specification or “executable spec”. The term “executable” implies that the model can be simulated to illustrate functional behavior. The term “specification” suggests an explicit design intended to meet certain requirements. In practice, it is common for the model alone to be called an “executable spec”, but this can lead to confusion and a loss of the intended improvement in productivity. In this work, we present the concept of a “complete” executable spec, which includes all the elements necessary to realize the gains in efficiency promised by model-based development.

DEFINING THE EXECUTABLE SPEC

An executable specification captures the functional behavior of a system. The authors contend that a complete executable specification must first include design documentation. There must be an executable model, and there must be a verification environment within which to run the model. Figure 1 captures these three broad elements, as well as the 7 deliverable components within them.

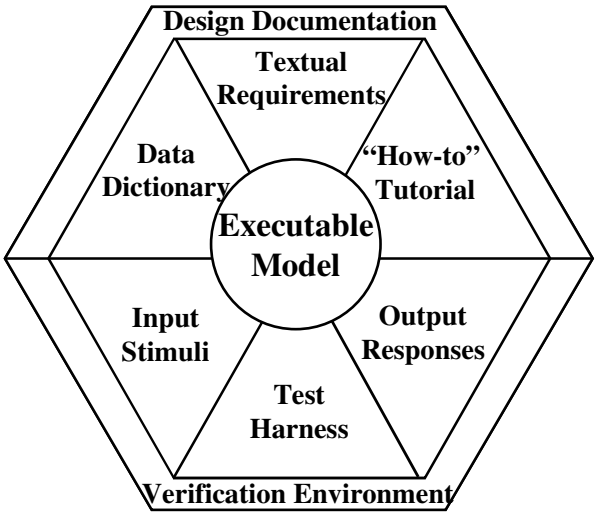

Figure 1. Graphical representation of a "complete" Executable Specification

The components of a complete executable spec are: (1) the text requirements describing the functions which must be met; (2) the math-based executable model; (3) a test harness to apply the inputs to the model and produce outputs; (4) the input stimuli or test vectors; (5) the expected or correct output responses to the input stimuli; and further documentation including (6) a "how-to" tutorial; and (7) a data dictionary. All of these components should be constructed so that a moderately-skilled practitioner can readily, and with little help, execute the model and be able to assess the output quality. In this way barriers to reuse are eliminated. For users who wish to use the model as a starting point for further development, this set of elements provides a clearly-defined starting point, as well as an example of a completed executable spec.

The scope of this paper is exclusively for the input processing, algorithms, control logic, diagnostics, data recording, high-level drivers, and communications required for electronic control units (ECUs). This portion of the system may be implemented as an application-specific integrated circuit (ASIC), or as software code in a general-purpose microcontroller. There are many other aspects of a system that are not included in such an executable specification. Color, size, corrosion resistance, connector geometry, and drop-shock resistance are important requirements, but are typically not modeled as a part of an executable spec. Certainly some of these characteristics can be modeled, and, at some future date, integrated with the logic and algorithms, but at present these are not considered.

The remainder of this section illuminates the central issues in the creation, use, and application of executable specs through a series of contrasts. Each pair of comparisons is loosely related to a "traditional" versus "model-based" development cycle. They also aim to clearly define certain terms which are often used in an ad hoc manner by practitioners of model-based work.

SPECIFICATIONS VS. REQUIREMENTS

A requirement is defined as a written document that describes what is needed without defining specifics of implementation. A specification is a written document that describes both what is needed and some of the details of how it is done. Complete model-based specifications are a more effective tool than traditional requirements mostly due to the ability to present mathematical and behavioral information in graphical and executable form. Table 1 shows a side-by-side comparison of the various elements of an executable spec between the traditional approach and the model-based methodology.

ELEMENT	TRADITIONAL	MODEL-BASED
Requirement	1. Non-functional 2. Functional "Thou shalt…"	1. Links to other documents 2. Given this input, expect this output.
Architecture	Teamwork, Visio or Power Point	Navigable, hierarchical, executable model
Design	Circuits, layouts, software, sensors and actuators, developed separately	Models of sensors, actuators, algorithms and plant or human-machine interaction
Documents	PDF, Word, Text	Auto-generated from the model
Verification	Bench test, in-vehicle testing	Progression from model-in-loop, software-in-loop, processor-in-loop, hardware-in-loop, human-in-loop, in-vehicle testing.

Table 1. Comparison of traditional versus model-based methodologies.

To realize the full efficiency benefits of a model-based approach using executable specifications, a relatively seamless suite of tools are required. Achieving straightforward process flow through the design cycle is a major factor in practical utilization of executable specs.

ARCHITECTURE VS. DESIGN

By architecture, we mean a representation of the partitioning, configuration, interfaces and hierarchy of a given product application. By design, we mean the completed implementation of the product application. Thus, architecture omits details to emphasize relationships. Ideally the design flows from the architecture during the successive implementation of details needed to make the architecture executable. When complete, the design is an instantiation of the architecture.

The value of a well-defined architecture to the development of complicated ECU systems cannot be overstated. It is well known that architectural decay is a constant risk as bugs are fixed, scope creeps, or requirements change – especially if there is pressure to complete these tasks quickly. Through the use of model-based design, the architecture is explicit in the layout and hierarchy of the model. With modular subsystems and well-defined interfaces the likelihood of architectural decay diminishes greatly. Complete executable specs, by nature of their test harness with inputs and expected outputs, allow one to perform regression testing whenever changes are made to the design. Furthermore, adding extra functionality can be a straightforward extension of the input and output set and update of the documentation. Systems designed in this way are easier to maintain, upgrade, and debug if needed.

FUNCTIONALITY VS. IMPLEMENTATION

In the traditional design approach, functional decomposition is used to sub-divide complex problems into more manageable components. The systems engineer thus partitions functionality and imposes hierarchy upon the design. The interfaces to each partition and to each level of the hierarchy require definition of the signal flow, the control flow, and the physical segmentation. This process proceeds in an iterative fashion until the individual components are atomic. Thereafter, the atomic subsystems are created by individuals or teams who must satisfy derived requirements as well as the interface definition. During this process, the systems engineer or product architect is selecting between competing choices or tradeoffs.

With a model-based approach using an executable spec, it is inevitable that the creation of the model will lead to an initial first order implementation of the design when describing the functionality. This approach to system design provides an engineer the capability to simulate and verify aspects of the functionality and implementation of the design throughout the development process. This ensures that the design choices are correct, assuming *a priori* the correct functionality of the sub-systems.

It is often said that the process described here should be free of implementation details. Yet, it can be argued that the interface definitions constitute a portion of the implementation details, albeit at a higher level than the atomic subsystem. Extending this argument further, we can see that even so-called atomic subsystems can be further reduced. Consider the example of an electrical hardware engineer, who designs a circuit block containing discrete elements such as a capacitor. He or she may specify the capacitance value and tolerance, and whether to use a tantalum or an electrolytic capacitor. The capacitor is an “atomic” component to the hardware engineer, however, to the capacitor manufacturer; the capacitor is a system of its own. The capacitor has electrode plates, a dielectric material, leads and packaging all of which must function together for this system to work properly in an ECU application.

The point of these arguments is to illustrate that what is called “design” and what is called “implementation” depend only upon the scope of the individual making the distinction. Thus it can be seen that the functional decomposition of a system is more than simply design because it also contains some degree of implementation detail. This justifies the relaxation of a specification being “implementation free” as discussed in the preceding section.

TESTING: PROACTIVE VS. REACTIVE

Verification is the process of assuring that the feature, function and/or attribute does indeed satisfy the requirement. In a traditional approach, system requirements are verified at the conclusion of a design cycle, typically after the atomic subsystems have been integrated up through successive levels of abstraction until the system is complete. In some cases, for example when delivery schedules are compressed, the subsystems are often verified after the design, implementation and build process, not during. The obvious drawback of this approach is the late discovery of higher-level errors in either the derived requirements or the implemented design. Figure 2 presents a graphical contrast between the traditional approach and the model-based approach.

In traditional development, the algorithm is a passive component of test environment: it executes the functionality-based input stimulus and produces the output results. The problem is that those input stimuli are developed to verify the correctness of the requirements. They do not typically cover all “abnormal” behavioral conditions. In model-based development, a model is an active component, and a source for the generation of model-coverage test scenarios. The combination of functionality-based input stimulus and model-coverage

based input stimulus create a complete test environment for rigorous validation of the model and verification of software implementation at each step of development.

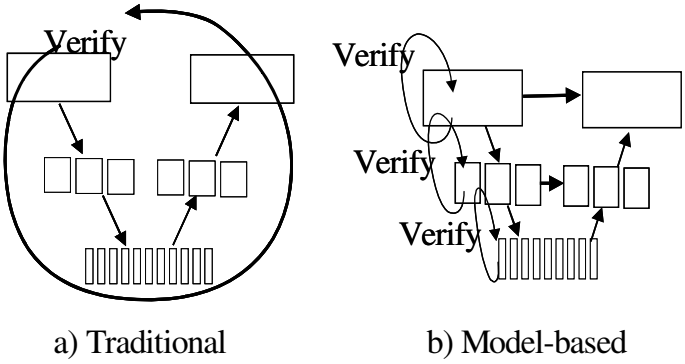

Figure 2. (a) Traditional V-chart for functional decomposition compared to (b) model-based development with concurrent engineering, early and rapid cycles of verification, and shorter design cycle.

PARADIGM SHIFT

ENGINEERING: SEQUENTIAL VS. CONCURRENT

In the traditional approach to system development, engineers are grouped by competency, typically hardware, software and systems, in order to create structured technical and procedural discipline. This organizational approach engenders a serial execution of tasks. For example, the system engineer must finish the functional decomposition prior to any other engineers beginning their work. Or, the software engineer cannot fully test their work until a hardware unit is available for bench testing. Such an inherent inefficiency is further exacerbated by the common occurrence that a system engineer may select design choices that may later be found to be sub-optimal from the standpoint of software and hardware architecture and/or design. Many small companies or small teams avoid this drawback by having multi-skilled individuals perform a wider range of tasks. Yet in larger companies, and for larger more complex projects, availability of such cross-skilled individuals may become a limitation. With model-based design, individuals skilled in system design, electrical design and software programming work in a common environment. This is the concept of co-design.

Co-design is the practice of parallel or simultaneous product development at multiple levels of abstraction across multiple engineering disciplines. For example, once a system engineer has an approximate high-level model created defining the product architecture, a software engineer can begin at once to add into the model the features of the software design. This stage will be followed by auto generation or hand writing of code, and unit and component testing. An electrical engineer selecting sensors, capacitors, or other components can insert models representing them into the system model to observe the impact on performance. An IC design engineer can work with the model to understand the requirements, and the model can serve as a golden reference model for the IC design and verification team. The model can also be used in a virtual product simulation for software test bench-like development prior to fabrication of the chip. As the program advances, more details are added, which improves the fidelity of the model for other engineers. In co-design, the system takes shape in a near-simultaneous manner.

Products designed this way are completed faster. Certain classes of errors are far less likely to arise when design is done this way. Many classes of errors that do arise are discovered early where the impact to the product development schedule is minimal. Working in parallel allows for a greater interaction among engineers, thereby leveraging teamwork benefits. Designing in parallel means shorter product design duration, so there is less time to forget details. Having three types of engineering discipline co-developing can dramatically impact performance metrics.

DEVELOPMENT: PROCESS VS. METHODOLOGY

For the purposes of this paper, process is defined as the steps to build a product. A methodology is how you choose to accomplish a given step of the process. We therefore speak of process design versus design methodology. Process explains what you do, and in what sequence. Method is how you do it. Ideally, a common process can be defined for any automotive ECU development process, and many companies have such a system in pace. Traditionally, methodology is often left up to the individual, allowing creativity but inviting proliferation and redundancy. The objective of the executable spec methodology is to apply math-based tools so that engineering work is more efficient and effective. The opportunity for creative implementation still exists, but the tools available help assure that the “how” is sufficiently powerful to design good products.

Applying the methodology of executable specs will manifest somewhat differently depending on the process to which it is applied. Vehicle systems with complex human-machine interaction (e.g. multi-media or navigation) are designed quite differently from autonomous embedded systems (e.g. airbag control module or powertrain controller). Yet, in each case there are certain broad phases of development, often captured within a common process or standard operating procedure. Also, from different product areas, different lessons are learned. Therefore, to understand what an executable specification is and how they are used, a number of disparate examples are required. In the next section seven case histories are presented, representing both a vehicle manufacturer working with a tier 1 supplier, as well as applications internal to a tier 1 systems supply house. By studying these applications and their post-mortem accounts, answers to the questions posed in the title of this paper can be obtained.

CASE HISTORIES – EXECUTABLE SPECS

CASE 1. EXECUTABLE ASIC SPECIFICATION

The semiconductor industry commonly uses functional models as a part of the IC design process. The use of functional models is more prevalent in SOC (system-on-chip) design, but they are often used in the design of complex ASICs as well. These models are typically written using C, C++, or system descriptive languages (SDL) such as System C [13]. The functional models are often referred to variously as executable specifications, behavioral models, algorithmic models or golden reference models. The models can be used to convey functionality and/or architecture of a design to a development and/or implementation team. They can also be used as a feedback medium to the author of a traditional textual specification as a means of confirming the proper understanding of the original design intention. If a functional model is properly developed with insight into the entire semiconductor design process, it can be used as a means of enabling early verification of a given design. In this case an environment is created to allow simulation and functional verification between the reference model and the multiple levels of abstraction that represent the actual design at various points in the ASIC design process. This environment can enable directed testing where the design is checked against known results. It can also enable indirect or random testing where the results are checked to ensure that the results of the actual design matches the reference model.

In an ideal world, the use of an executable spec could enable substantial benefits in an ASIC development process at the semiconductor supplier as well as at the system design house. The most significant benefits being: upfront specification clarification at the start of the development process; better functional requirements communication; early software development; and, a more complete verification process. Of course, these are all not realized to their full extent without expending engineering effort in bridging the system and the IC design environment.

ASIC Project

Figure 3 shows a graphical representation of an executable specification for a custom, mixed signal ASIC developed for use in an airbag deployment module. Three levels are overlaid: the graphical user interface (GUI), a block diagram defining the architecture, and the circuit layout using standard cells. The development of this ASIC consisted of the integration of two existing mixed signal ASICs plus additional features and functions. In this case study, the development team created a virtual prototype of the ASIC directly from the system requirements and independently from the ASIC design team.

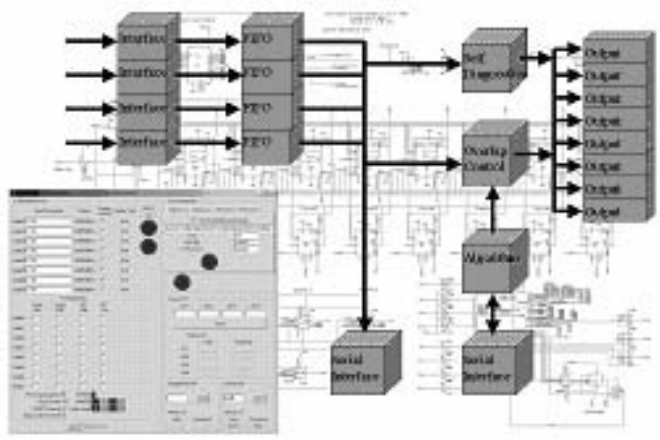

Figure 3. Graphical Representation of an Executable Specification and Associated Test Environment.

This executable specification was developed as a test case for communicating the intended functionality of a design to an ASIC supplier. The executable specification was delivered to the supplier as three parts: a textual specification; a C language functional model; and a test environment which includes input stimuli, output responses and documentation. Product-level development teams, as well as the ASIC design team, used the executable specification to augment their existing development process. The model was also written in a format that allowed compatibility with a cycle-accurate system level simulation environment where the production software used to configure and control the ASIC was developed and verified.

At the semiconductor supplier the executable specification package (model, test environment and specification) was used to implement the ASIC design. The test cases delivered in the executable specification, in addition to some of the system level software code, were manually re-written in a register transfer level (RTL) language for compatibility with the ASIC design and verification environment. In both cases, the re-written code was used to verify the functionality of the ASIC design [12].

Lessons Learned

As a result of this project, more than thirty specification clarifications were identified and resolved early in the design process. Three major functional issues were identified, the first of which resulted in a design change while the remaining two issues resulted in specification changes. The use of an executable specification reduced the number of silicon turns on this project by one, saving perhaps 4 months and avoiding considerable charges for tooling and IC fabrication. The use of the functional model allowed the software and systems design teams to start their development effort six months prior to receiving the first pass silicon from the supplier.

Going forward, it has become clear that more work is needed in bridging the gap between the system level verification environment and the ASIC design and verification environment to realize the full power of this methodology. A more seamless and automated approach needs to be developed to ensure accurate and efficient translation of test cases. Overall it was shown that the use of the executable specification in a high level graphical simulation environment could enable earlier software development and earlier software verification as well as improved IC design results.

CASE 2. ROLLOVER ALGORITHM

Sensing vehicle rollover requires, at a minimum, an angular rate sensor. An algorithm to discriminate rollovers from non-rollover events was developed internally by a team of five engineers. The algorithm was implemented as a model and thoroughly verified over a wide range of rollover test events.

After verification, the floating-point model was converted to a fixed point implementation and autocoded. The autocoded software was then ported to a microprocessor evaluation board which provided metrics on memory usage and throughput. Immediately evident was a very large usage of random access memory (RAM) and a spike in throughput during certain times of operation. To reduce these values, a finite impulse response (FIR) filter was replace with an infinite impulse response (IIR) filter, along with a slight modification to the algorithm logic. This change reduced both RAM and throughput by approximately 75 percent [6], as shown in Figure 4.

Integrating the autocode within the rest of the product software required approximately 12 hours of engineering effort. The main body of the autocode was mostly un-changed, with one exception being the need to move from 16-bit to 32-bit variables in the IIR filter to preserve accuracy.

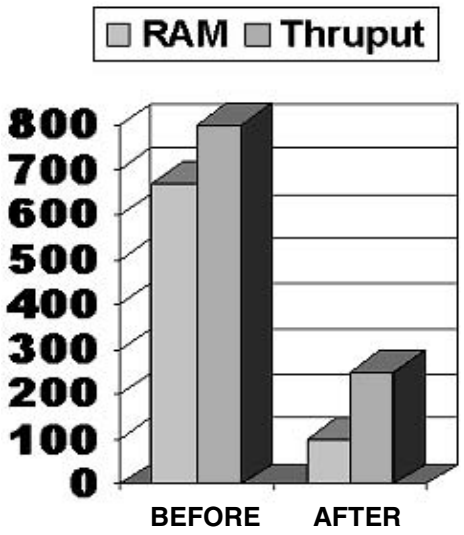

Figure 4. Improvements in RAM (in bytes) and maximum throughput (in microseconds) for a rollover algorithm through the use of an evaluation board.

Lessons Learned

The system engineers developing the algorithm were not well-versed in fixed-point math. Yet the autocoding tools make this process intuitive and fast. Of great benefit is the rapid feedback on microprocessor resources, since this information may sometimes takes months as hardware is built and software is written. The simulation environment created for the original FIR model was instantly applicable to the IIR model, so the re-verification took less than one day's work. The autocoded rollover algorithm is currently in production.

CASE 3. RADIO CD PLAYBACK

The CD playback mechanism is an important part of modern radios. With many suppliers there are many playback mechanisms to choose from, with new developments occurring each year. Each new playback deck requires interface routines that process user inputs (button presses) and determines the proper commands sent to the deck. In addition to implementing the desired behavior, error handling for faulty decks or imperfect CDs is a significant portion of this layer of control software.

Figure 5. Car radio faceplate. The CD slot is at the top.

For this project no requirements existed. Therefore the first step in development was to gather relevant documents and interview key practitioners. A formal architecture did not exist for the playback control layer, so this was created as a framework within which to develop the behaviors and error handling. Hardware-in-the-loop (HWIL) testing is crucial for verifying the software, so the capability to interface executable specs to a serial message bus interface was also created.

The completed project developed a significantly faster method of preparing new playback decks for production. From a reuse library, behavior and error handling routines can be added to the architectural framework to build the application quickly. Test harnesses have been created which allow rapid unit testing. Model coverage tests were done to ensure that all paths are reachable. Autocode generated from the model was placed back into the simulation environment, which was then used to exercise the playback deck. The complete HWIL setup time for converting to a new deck can now be completed in hours or days, instead of weeks or months.

Lessons Learned

Important lessons learned on this project were the limitations of the current version of tools used for easy implementation of a product's architectural ideas. Although temporary workarounds are suitable for development, a more streamlined approach is needed in a production setting, and we are working with tool vendors to address these needs. Another valuable lesson is the much greater understandability afforded by a hierarchical model-based design. New engineers brought into this project were able to quickly learn how and where to make contributions, how to test and verify their portion of the design, and how to place their work into the reuse library using the concept of a complete executable spec as defined above. Configuration management (CM) was applied to all models, libraries and tool scripts. We discovered that using CM tools required slightly different handling compared to simply using documents.

CASE 4. VISION ALGORITHM LIBRARY

Vision-based pattern and object recognition can be a difficult problem. Several archives of machine vision routines are available to perform commonly-used operations such as histogram equalization, edge-detection, or image binarization. However, these routines are typically written in C++, and are not directly suitable for implementation in low-cost automotive microprocessors. Bridging the gap between advanced development and production implementation involves porting the application to the C language. Instead of repeating this onerous task another approach is being used where executable models of commonly-used routines are created and placed in an on-line library. In this way, both researchers and production teams can use the same library, and the auto-generated code can be ported to any target microprocessor.

Lessons Learned

This project is on-going, but has already provided insights into important aspects of this use of executable specs. Because vision algorithms are usually in C++, researchers are often fluent in this language. Compiled C++ code runs fast (compared to a model), so researchers are reluctant to change tools. However, the benefits to the product team are considerable, and autocoded vision algorithms have been downloaded to a digital signal processing (DSP) chip and run real-time in a vehicle application. This process took just days, compared with the C++ to C conversion which can take weeks. As more users adopt the model-based library approach, further benefits are expected in the time between patents and products.

5. BODY COMPUTER

A leading European automaker initiated a project to evaluate requirements using executable specs. This body computer application is responsible for wiper/washer mechanisms on the front and rear windows of the vehicle. The goal for this project was exclusively to validate the requirements. Towards this end, a graphical user interface (GUI – see figure 6) was created to allow convenient interface to the executable model, and a batch test environment was created for rapid evaluation of multiple scenarios.

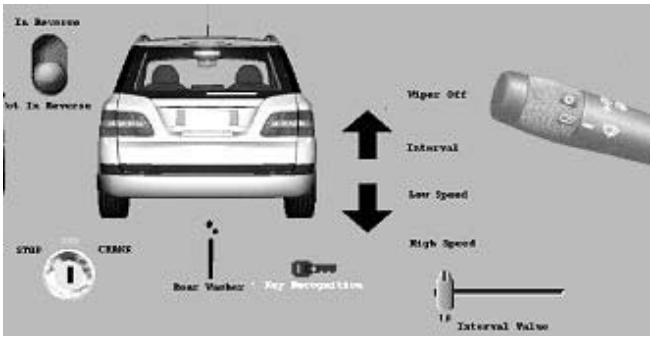

Figure 6. GUI for a body computer controlling wipers and washers, front and rear.

Lessons Learned

The primary focus for this project was to validate and complete the textual requirements. Making a clear distinction between requirements and specifications is the primary lesson learned. The executable model is a high-level implementation of the requirements which can be tested and refined early in the development cycle. Although autocode was not part of this project, the executable model of the wiper/washer was used as a specification to the software engineers who then wrote code by hand.

CASE 6. ADAPTIVE CRUISE CONTROL PROJECT

The specification for the software implementation of the adaptive cruise control (ACC) algorithm were provided by the customer in a form of executable model implemented in Simulink/Stateflow. The tier1 supplier was responsible for autocoding of the model and up-integration of the coded software with the product level software application.

After the original evaluation of the specifications, two additional artifacts of executable specs were requested and received from the customer: requirements documentation that contained a data dictionary; and a simulation environment that contained vehicle data, which was collected by the customer during rapid prototyping and served as the suite of input stimuli and output responses. In addition, using the executable spec as an active component, model-coverage based test vectors were automatically generated using a commercially available software tool.

Project Benefits

During the lifetime of the ACC software project, the supplier received more than a dozen versions of the model. By applying autocode technology, the supplier was able to reduce turn around time for the fixed-point software implementation by 40% compared to traditional handcode based software development. Quick implementation of the changes in the algorithms allowed the customer to check more variations of the algorithm on the vehicle fleet and select the best one. The memory (~80K) and the throughput consumption of the autocoded software were on par with the handcoded implementation of the similar algorithms. The autocoded software was released to production.

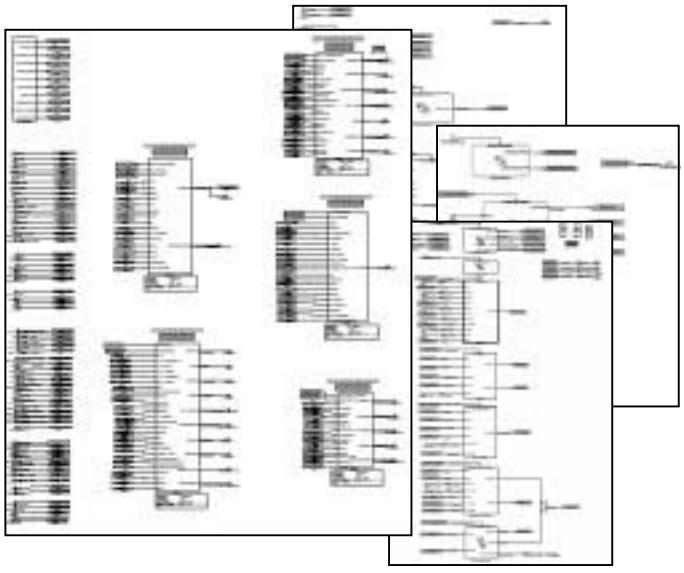

Figure 7. High-level view of the design layout for the adaptive cruise control project. Selected subsystems are displayed to the right.

Lessons Learned

The use of executable specification and autocode for the production project involves the creation of the development environment that accommodates changes in the specifications and autocode in an easy and repeatable way. It dictates the existence of supporting tools like configuration management, model differences, model merging, and a universal data dictionary. Currently not all these tools are available on the market. Therefore, additional time should be reserved in the production schedule for the development of the supporting tools. Secondly, the modeling and autocode guidelines should be developed prior to the start of a large project. Both engineering teams, the algorithm developers and autocode software engineers should follow the same common guidelines. If will help assure the maintainability of the model and efficiency of the generated code.

CASE 7. NOTIFY VEHICLE OP (SEAT BELT REMINDER)

The National Highway Traffic and Safety Administration (NHTSA) within the US Department of Transportation regulates that vehicles sold in the US have a seat belt reminder (SBR) system, also called “notify vehicle operator”. This functionality was implemented as an executable spec and had three primary objectives:

1. Assess and implement the SBR requirements received from a vehicle manufacturer.

2. Generate autocode for production.

3. Perform verification testing to the requirements as well as worst-case scenarios.

The GUI created to interface with the state diagram of the SBR system is shown in figure 8.

Project Results

The verification plan included a total of 75 scenarios which were generated and run across the model. Results were captured automatically in a spreadsheet for review. A total of seven requirement discrepancies were found through the verification process. Near the end of this project, the customer changed the system requirements in several areas. The needed changes were made and verified within 2 days. The final model was placed in a re-use library for future development work.

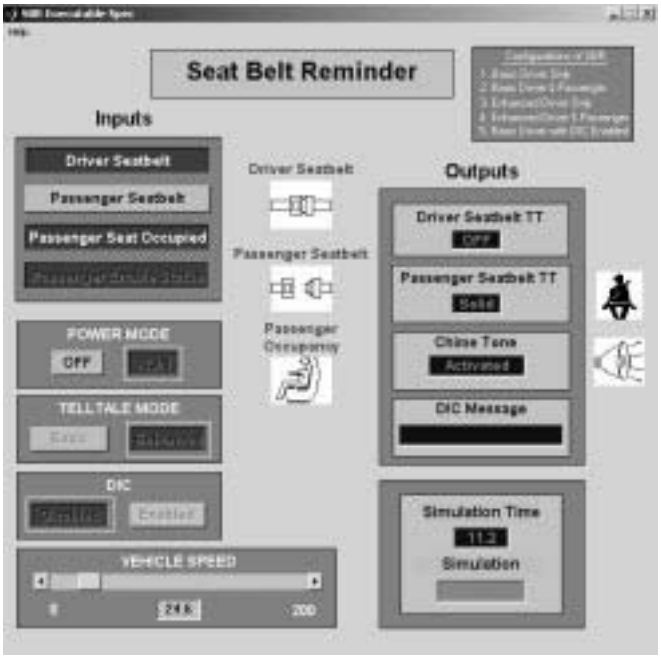

Figure 8. GUI for Notify Vehicle Operator, giving access to configuration settings and system inputs. Outputs are graphical, visual and auditory, and results can be saved in text format.

Lessons Learned

The testing performed on the SBR functions allowed the discovery of requirement errors very early in the project. Both the interactive real-time GUI and the off-line batch simulation capabilities provided the ability to test a wide range of scenarios, even going beyond the requirements, without adding much time to the project completion.

Autocode generated from the model was reviewed, but not used in production. The product team did not have sufficient confidence to use the autocode, even though the host-compiled code was directly tested against the same scenarios as the model (golden reference). Introducing new implementation methods within the engineering community commonly meets with resistance. Even when clear benefits in accuracy, thoroughness and cycle time are evident, the natural human inertia to change, and inherent skepticism towards new ideas, must not be ignored.

BENEFITS OF MODEL-BASED DEVELOPMENT

PRIMARY BENEFITS

A. Specification Discovery – language-independent system behavior can be demonstrated to global customers, clearly conveying the expected performance.

B. Golden Reference (Single-source of information) – a central golden model can be used for documentation, calibration, source code, and interactive graphical user interfaces, plus regression testing when changes are made.

C. Architectural Exploration – using either cycle-accurate simulation (IC example), a microprocessor evaluation board (rollover example), or HWIL (audio example) along with reuse libraries, design choices such as hardware/software partitioning can be evaluated to determine resource needs, allowing early and accurate estimates of manpower and components.

D. Design Quality – automated checking catches certain errors, and the visibility of the architecture supports a methodology conducive to high quality designs.

E. Verification and Testability – models allow for batch testing at many levels of details, so that many more test cases can be explored compared to traditional bench or field testing. The functional test cases are used (sometimes with added detail) for the model as well as the fully implemented product.

F. Autocode – tools to generate embedded code from executable models avoid certain coding errors, reduce development time and help ensure congruency throughout implementation.

SECONDARY BENEFITS

G. Co-simulation tools can be interfaced so that each element of the design process can be developed using other elements as a test bed.

H. Continuity – the use of models from concept to development to production reduces waste and shortens the time from patents to products.

I. Graphical Representation – many modeling tools allow for graphical system creation, facilitating ease of understanding and communication.

DISCUSSION

The traditional method to capture customer requirements is a written document. These written specifications typically contain charts, tables, graphs, and diagrams in addition to the textual language to convey product or system specifications. In some cases, important examples might be included to ensure the desired functionality and performance. What follows is a summary of the sources of inefficiency in a traditional approach:

1. Written requirements are subject to misinterpretation, and derived requirements

may not anticipate all interactions between components.

2. Testing at the end of integration implies long and repeated cycles of learning, inability to adapt to changing needs and severe impact to time-to-market.

3. Weaknesses in the specifications tend to be discovered late in the process, and delivery pressures often favor the quick fix.

A graphics-based executable model with the seven elements of completion described above allows full realization of the benefits of model-based development.

CONCLUSIONS

Math-based development through the methodology of executable specs can be applied to a wide variety of applications. While the tools used may change, and the process flows may have different emphases, we have seen improvements in time-to-market, product quality and overall development cost. The examples above illustrate some of the details needed to understand how executable specs can be used, where to expect benefits, and how to understand the limitations.

Executable specs do not solve all problems. Good engineering practices are essential. An informed management is essential. Benefits may not be realized on the first introduction due to the front loading and learning of the design process. A disciplined, methodical campaign of training, mentoring and execution is needed for implementation, sustained by long-term vision.

In order to gain the maximum entitlement to using executable specifications, engineering organization will need to start shifting away from sequential development flows toward concurrent development flows. Engineering skills need to broaden across the typical boundaries of hardware, software and systems. Design methodologies will need to augment the standard development processes already in place. This combination will ensure that both the “what’s and the how’s” are defined for developing new products using executable specs.

Further consolidation of math-based development will eventually see greater coordination with electrical design, EMI modeling, package process and tolerancing, test development and artificial intelligence optimization. Through early adoption of math-based methods, an engineering organization positions itself to reap benefits in efficiency and design quality as further technical development become available. Those companies aggressively developing these capabilities now will be those which prosper in the future.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge contributions to the content and intent of this paper from Randy Brunts, Everett Lumpkin, Michael Gabrick, Rick Nicholson, Nambi Ganesh, David Braun, Chad Aeschliman, BC Manjunath, Jason Molenda, Branislav Kisacanin, Imran Ahmed A, and Manoj Dwivedi.

REFERENCES

1. “Model Based Systems Development in Automotive,” M. Mutz, M. Huhn, U. Goltz, C. Kromke, SAE World Congress 2002, paper 03B-128.
2. “Zero Hand Coding Approach for Controller Development,” G. Saikalis, S. Oho, S. Zunft, “SAE 2002 World Congress, paper 2002-01-0142.
3. “Trends of Future Powertrain Development and the Evolution of Powertrain Control Systems,” T. Ueda, A. Ohata, SAE Convergence 2004, 2004-21-0063.
4. “Model-Based Tools Update,” The Hansen Report on Automotive Electronics, June 2001, vol. 14, no. 5.
5. “A System-Design Methodology: Executable-Specification Refinement,” D. Gajski, F. Vahid, S. Narayan, European Design and Test Conference, 1994 Proceedings, March 1994.
6. “Systematic Model-Based Testing of Embedded Automotive Software”, M.Conrad, I.Frey, S.Sadeghipour, Electronics Notes in Theoretical Computer science 111 (2005) pp.13-26.
7. “Incorporating Autocode Technology into Software Development Process”, L.Vitkin, T.K.Jestin, ICSE 2004, pp.51-57.
8. “Managing the Challenges of Automotive Embedded Software Development Using Model-Base Methods for Design and Specification”, M Yeaton, SAE 2004-01-0720
9. “Integration of the Code Generation Approach in the Model-Based Development Process by Means of Tool Certification”, I.Sturmer, Journal of Integrated Design and Process Science, Vol. 8(2), pp.1-11, 2004.
10. “Automotive Software development: A model Based Approach”, M.Rappl, P. Braun, M.von der Beek, C. Schroder, SAE 2002-01-0875.
11. “Design and Implementation of a Rollover Algorithm in Production,” P. Schubert, dSPACE User Conference, 2004.
12. “Design Process Changes Enabling Rapid Development”, F Winters, C Mielenz, G Hellestrand, Convergence 2004-21-0085
13. “Reuse Methodology Manual For System On Chip Designs”, M. Keating, P. Bricaud

Peter J. Schubert, Ph.D.
Packer Engineering, Inc.
1950 N. Washington St.
Naperville, IL 60566-0353
630-505-5722 or 800-323-0114