Scott Ranville | *"The Software Beret"* | 1227 W. Weaver Cir.
work: 303-734-8988 | | Littleton, CO 80120
cell: 303-931-3070 | State-of-the-Art Embedded Software | www.softwareberet.com
fax: 801-457-9698 | Tools and Processes | scottranville@softwareberet.com

Improving Deeply Embedded Control Software Development Processes

By: Scott Ranville
The Software Beret
Oct. 2002

1. Executive Summary

This article will tackle the question: "What does a company need to be doing in order to have a competitive software development process for deeply embedded control applications?" The general ideas talked about in this article apply to any model-based process, but the specific tools that are mentioned only apply to a Matlab-based process. Also, my background is automotive, so this will be weighted towards automotive powertrain applications.

A number of tools (automatic modeling style guide checker – MINT, on-target rapid Prototyping – RapidHawk, automatic code generation – TargetLink, automatic unit test vector generation – Reactis) exist today that can be introduced into the production software development process that will improve the quality of the generated code, while lowering the overall costs and development time. There are a number of other technology areas which would benefit from tools, but for which the tools do not exist today. Each company needs to position itself so that when these tools become available, they are in a position to take advantage of the tools. It is also beneficial for each company to work with the various tool vendors and research groups to direct how the tools and research evolves such that the end product will be able to be used within the your company's production software development process. I have done this for a number of tools. Before the tool vendors incorporated my recommendations, the tools did not solve the problems of the end users and would not fit into their development processes. After working with me, the tools have gone from the research groups in the companies to the production groups who are in the process of making them part of the production software development process. Thus, without end user input, these new tools will not meet the end user needs.

This article will describe the long-term process objectives, describe the author's view of today's process, and finally describe the desired software development process.

2. Long-Term Objectives

When defining long-term process goals, it is necessary to define what the long-term objectives are and then to figure out how to achieve this long-term vision. This section will define these long-term objectives.

The first key long-term objective is to automate wherever possible. In today's process, there are a number of manual steps each of which introduce time and are sources for errors. With the technology that is available today, a number of these manual steps can be partially or fully automated. This automation will result in shortening the overall development time while at the same time producing a higher quality end product. The most promising automation opportunities today for a Matlab-based process include style guide compliance checking and correcting, on-target rapid prototyping, automatic code generation, and automatic unit test vector generation.

Unfortunately, today's tools do not allow a single model to easily go through all the process steps with the associated tools. One way to address this process flow concern is with a modeling style guide. The style guide can be designed such that unsupported modeling constructs can be avoided. The style guide can also require "aids" to be added to facilitate other automation scripts. These "aids" may be as simple as a character string that is used within a field in a library block. A script that indicates to the tool that the block needs special processing can then easily identify this character string. Checking the compliance to the modeling style guide is a manual process that can be time consuming and error prone. An automated style guide compliance checker can perform many of the needed checks in a matter of seconds instead of the previous hours that a person may have taken. An additional automation step that can enhance the style guide checker is a style guide fixer that will automatically fix "simple" style guide violations. Thus, the style guide checker and style guide fixer will reduce the overall development time while allowing a model to more seamlessly proceed through the software development process.

Rapid Prototyping is one of the process improvement initiatives that have been evolving over a number of years. This provides feedback about the correctness and effectiveness of a control algorithm earlier in the development process than if the control engineer had to wait for the software specification to go through the rest of the software development process to get coded and then tested on the final application. Thus, the benefit of Rapid Prototyping is early feedback on the control algorithm. Until recently, one limitation in this feedback is that a high-end processor has been needed that is considered to have "infinite speed and memory resources". While this type of system does provide benefits, it does not answer the important question of whether or not the algorithm will work on the production microprocessor that does have speed and memory constraints. The on-target rapid prototyping takes advantage of the improving autocode tools to generate C code that will fit on the production microprocessors. This code does not meet all the production requirements, but does allow the controls engineer to get very close to a production level of code in which to test the algorithm. The on-target Rapid Prototyping is a mechanism that allows for better software requirements to be designed early in the development process, thereby reducing design errors, shortening the development time, and reducing overall costs. As reported in "Removing Requirement Defects and Automating Test" by Mark R. Blackburn, Robert Busser, and Aaron Nauman, "Boehm and Basili … state that finding and fixing a problem late in the development process can be 100 times more expensive than finding and fixing the requirement or design phase."

Another of the manual, and therefore, error prone steps is generating code from the software specification. Recent advances in automatic code generation technology allows for software specifications, which are in the form of models, to be automatically converted to efficient C code. Depending on the autocode tool, this code can be as efficient as hand generated code, and in some cases even more efficient than the hand code. Some of the autocode tools have also added enough flexibility to allow the autocode to satisfy company specific coding standards, for example, many companies have special rules about which files variables are declared in. These company specific requirements are needed to allow the autocode to be used with the large base of legacy that the automotive companies will be using for a number of years to come. Automatic code generation will shorten the software development time because it will take the tool a few seconds to generate the code where as it takes a person a number of hours or even days. However, a bigger benefit will be the reduced number of errors that are introduced by converting the software specification to code. Lastly, another minor benefit will be that once the code generator is trusted, code inspections will not be needed to check for compliance to coding standards and translation errors.

The introduction of models into the production software development process is a new process step. Many of the models are being created by converting existing C code into a model (as opposed to creating the model from scratch or from another "requirement" format). As with all manual steps, the converting of the code to a model is an error prone step. In most cases the C code is trusted and the question arises as to whether or not the model functionally matches the code. A new tool technology, automatic unit test vector generation, will generate the input stimulus that can be applied to both the model and the code. If the model and code outputs match for all time steps, then there is a certain degree of confidence that the model and code functionally match. The degree of confidence depends on the type of test vectors that are generated. Today's tools generate test vectors primarily to test the control flow logic. Some common coverage objectives are statement coverage, branch coverage, and MCDC coverage. The MCDC coverage level is an FAA requirement and does provide a high degree of confidence that the control logic within the model and the code functionally match. Because the unit test vector generation tools are so new, a good quality control step at this time is to use model and code coverage tools to measure the model and code coverage levels that are actually achieved. This will provide quantifiable measures that the model and code have been tested to the same level. For example, if the coverage levels do not match, then closer inspection is needed to see if the model has extra or missing components as compared to the code. Because automatic code generation is not yet part of the production process, the automatic unit test vectors can also be used to improve the confidence that the hand generated code functionally matches the model. Thus, the unit test vector generation technology will help identify translation errors earlier, thereby providing for a higher quality end product while reducing overall development time and costs. A reported in "Specification-Based 'Safe Code' Software Development Using BEACON" by Stephen Morton February 21, 2001 "The Hughes Composite Software Error History reports that a bug caught in `unit testing costs as little as one tenth with respect to that same bug if caught during post-release maintenance."`

The second long-term objective is to avoid all manual translations between different tools and different process steps. As mentioned above, all manual steps involve time and are sources for error introduction. To try to catch these errors and prevent them from progressing through the rest of the process, most groups add "quality control" steps to the process. These quality control steps are often manual steps and involve "too much" time. One way to address these manual translations is by selecting a "core" modeling tool which all the other process steps must use. One of the more popular automotive "core" modeling tools today is Matlab, but even this tool does not allow support for the entire software development process. A "core" modeling tool will allow for a smoother flowing process with fewer translation errors and will reduce the number of quality control steps that are needed. This will reduce the overall process development time and result in a better end product.

The third long-term objective is to avoid all duplication of information. Multiple sources of information take time to create and it becomes very hard to keep all the sources consistent. As an example, many different process steps need to know the variable type. Instead of entering this information in each tool separately, a central data repository should contain this information, and a 'model preparation' script should extract this information from the central repository and convert into the format needed by the particular tool. As a second example, a number of modeling constructs get reused in the models. Instead of the engineer recreating this construct each time it is needed, there should be a library block that the engineer can add to the model. As a last example, Matlab supports adding a label to a signal line, and also supports the propagation character, "<". This propagation character allows for the signal source to contain the desired label. Then, the tool will automatically add the signal name to all subsequent signals that use the propagation character. Thus, for overall efficiency improvements, consistency, and to reduce maintenance time, there should only be one source for all information.

The fourth long-term objective is to have all relevant information formally documented. When trying to automate the process, information that resides in the engineer's mind cannot be automated. For example, if the variable's type is in the engineer's mind, the "model preparation script" will not be able to automatically convert this to the format needed by the tool. As another example, a second engineer that is developing another feature that uses this variable will not be able to find the variable definition and may implement it differently. Thus, the benefit of documenting all relevant information is that the tools can access this for the desired automation. Also, if the engineer leaves the company or moves to another position, their knowledge base is not lost. This will result in a more robust overall process and in a higher quality of the end product.

The fifth long-term objective is to reduce the wall/chimneys that exist between different groups. The groups need to work together to achieve an end product that is optimized instead of trying to optimize their area of responsibility. In order to achieve the optimal solution, system level tradeoffs are needed. This means that some groups will not have an optimal solution for their particular area of responsibility. As an example, automatic code generation will allow for the hardware present and not-present calibration parameters to be set before generating code. Then, during code generation, the autocode tool can only generate the code that is really needed. This may mean that "build" engineer has to define the present and not present hardware sooner in the software development process. The "build" engineer will also have to take care to use the correct version of the code. Thus, the "build" engineer may complain that his job has been made more difficult, but the reduction in ROM and RAM needs may lower the hardware costs and be a net gain for the company. (For those not aware of this, hardware costs are a huge deal in the automotive world. Using off-chip memory can cost a program millions of dollars over the life of the vehicle line.) If the build engineer's manager will authorize the development of another automation script, one can be developed that will partially automate the build engineer's job and end up making the job easier. Thus, by eliminating these walls and working together, the final product will be of higher quality, lower overall cost, and therefore improve customer satisfaction that should increase market share.

3. Today's Process

In order to propose process improvements along with steps that are needed to get to this vision, it is imperative to understand the existing process. This is required as most groups will not accept radical changes to their day-to-day work, but require smaller incremental changes. Also, management needs to feel confident that the changes will indeed benefit the overall process and still let them meet the production delivery schedules. This is best done with gradual changes in which each step shows success before implementing additional changes. Non-automotive groups would argue that this is an inefficient way to implement process change, but a key factor for the automotive environment can be termed "software factory". Within automotive, there is a high degree of software reuse. A typical "new" application will take a previous application and touch maybe 20% of the functions. Touching a function may be as small as a one line code change. Thus, "clean sheet" projects are very rare. With this software factory there may be a major application release every other day. As with most industries, the production engineers are overworked, and do not have the time or luxury to get re-trained and take a big chunk of time to convert their 'feature' to a new tool or format. Thus, incremental changes with immediate, quantifiable benefit are needed to allow the production deadlines to be met and be acceptable to the working engineers and management.

This section will establish what the current production software development process looks like to an outsider. I use to be a Ford employee, but I was in research and was considered an outsider by the production engineers.

The first step is to collect requirements on what the application is suppose to do. Within automotive, there are a few sources of "requirements". A predominant source is from the 'calibrators'. These are the engineers who do the in-vehicle testing and "fine tuning" of the vehicle performance. Often these requirements are really bug fixes with some algorithm improvement suggestions. Some other, more traditional, requirement sources are government regulations such as emissions and marketing requirements for improved customer satisfaction. The requirements capture does receive extra attention when defining a new application, but, for the most part, is on ongoing process which non-automotive industries may call requirements creep.

The requirements definition is followed by "architecting" the requirements, which will establish major divisions and the resulting interfaces. For example, will a single microprocessor be used, or a multi-processor implementation. As with the requirements, prior architectures are often reused. This step may find some requirement contradictions or missing or incomplete requirements. If found and fixed at this stage in the process, this will result in the lowest cost and quickest solution. This architecting is typically done by the same group that collects the requirements, with some additional technical specialists being added to the team.

Once the requirements and architecture have been established, the individual 'features' (sometimes called components, units, functions, sub-functions, rings, etc.) can be developed. These features need to follow a modeling style guide so as to facilitate the model being used as seamlessly as possible throughout the rest of the process. At this time, requirement and architecture clarification and refinement continue. Once developed, the models need to be validated that they indeed meet the original requirements. A number of validation techniques exist such as control theory analysis, simulation, rapid prototyping, and HIL. Ideally, all requirement and interface issues will be resolved at the end of this step, because once these errors get past this stage, the cost and time to find and fix them starts to go up significantly. Typically an individual engineer or small group of engineers develops each feature. This same engineer is responsible for validating the model.

After the model has been validated, or the development time expires, the model gets reviewed by a model review board. This is a quality control step that is the last effort to prevent future costly requirement, architecture, and algorithm errors. The board checks for compliance to the modeling style guide, checks for compliance to the defined interfaces, and that the original requirements are indeed satisfied.

After the review board has "ok'd" the model, the model documentation can be created. This documentation will then be used by the software coders as well as the calibrators for the final application testing and fine-tuning.

Once the documentation is ready, the feature coders start writing the final production code. While not always done, these coders have the option of testing the software on the PC and finding and fixing the first wave of errors. Doing this will reduce the cost of finding and fixing the errors by a factor of 10. As the last quality control step, the code gets inspected by a code review board. The code review board is checking for compliance to coding standards, translation errors, and also that the original requirements have been met.

Ideally, the architecture team will decide on final hardware and define all the needed low-level driver and RTOS/Scheduler features at the same time that they are defining the other interfaces. This will allow the "low-lever driver" coders and the RTOS/Scheduler coders to develop the needed code in parallel with the algorithm development, validation, and coding. In an optimized process, the low-level drivers will be available very early in the process. This will allow the feature engineers to do on-target rapid prototyping in their validation process. The RTOS/Scheduler is not needed quite as early the process, but could be used for the validation efforts and also for the integration testing.

Once the code inspection has passed, the software is released to the 'build' engineer. The build engineer is responsible for collecting compatible versions of all the features, low-level drivers, and RTOS/Scheduler that when grouped together will meet all the requirements. The build engineer compiles the code and releases the final application for in-vehicle and dyno testing.

After the application is released by the build engineer, the calibrators begin the final testing and "fine-tuning" of the software. The testing is the last effort to catch coding errors and requirement errors before the application is released to the end customer. The fine-tuning process sets the values of various parameters in an attempt to optimize the vehicle performance. To optimize reuse, the feature development includes a number of tunable parameters. Some of these include indications for hardware that is present or not present. Other parameters directly affect the performance, for example, gain values in a PI control loop.

For all of the above, configuration management is needed to keep track of the various versions of each file that are created. This allows for a controlled approach to changing the files and allows for previous versions to be reused if needed.

4. Desired Process

This section describes a software development process that will place the in-vehicle software development process among the leaders in the automotive industry. I am not going to claim that it is the best, yet, as I do not know what all the other companies are doing. From what I do know*, many other companies are considering these types of solutions and some companies have already successfully implemented some of these proposals. Thus, to not take a serious look into these will put your company at a competitive disadvantage.

> *This knowledge of other companies comes from conference papers and from talking to the tool vendors about requests that they have received for new features. Some of the tool vendors also publish tool user lists and even testimonials and case study results.

The capture of requirements needs to become more formalized and move away from being dominated by calibrator error reports. Today too many of the requirements involve "hand waving" and the requirement that "that is the way we have always done it". The requirements need to be written down and cataloged. This will address the problem of how when individuals change to new jobs that core information is lost. This will allow requirement tracebility and impact analysis to be conducted that will improve the final product as original requirements will not be missed and when requirements become obsolete, they can be safely removed. Also, when a change request comes in, more accurate estimates can be made for how long it will take to fix it and all the locations that need to be fixed.

The new desired process will have tools to allow the architecture decisions to be documented and tested. The architecture team will be able to analytically test and evaluate different options instead of having to extrapolate and guess at the implications of their decisions based on prior experience. This analysis will include such items as schedulabilty analysis, resource availability, and interfaces both within a single processor and between multiple processors. For the schedulability part of this tool, some "sub-tools" are ones that will generate 'time' test vectors that will force the path through the code that takes the longest time. Another "sub-tool" is a timing accurate simulator that can be used to measure the worse case execution time for the set of hardware settings that are being used.

A new element to the desired process is the need for a "central repository". With the increasing time pressures, need for higher quality solutions, and increasing complexity of the applications, duplicate sources of information and missing information cannot be tolerated. The central repository will provide one source of information that will get reused by in a number of the process steps. For example, a variable's type will be needed for typed simulations, on-target rapid prototyping, documentation, automatic code generation, and unit test vector generation. Instead of manually duplicating this information in all the different tools, a script can extract this information from the central repository and automatically convert it to the format needed by each tool. A properly implemented central repository will make the process flow smoother and result in a higher quality end product. As an example of what can happen if this is not done, the $7 billion European rocket, Ariane 5, exploded during take off because of a type error that had made it pass all their testing (total development time was a decade). Also, from an internet article, "NASA recently lost Surveyor 98's polar lander due to a software error. A peer review team found that one software development team used English units and the other used metric units in Earth-based mission software. The software failed to convert between the units and was critical for successfully navigating the lander into Mars' atmosphere, where it is assumed to have been lost."

The creation of the models will be made easier by scripts and extended library blocks for commonly used constructs. Not only will this speed the model creation time, but will also result in higher quality models as there are fewer places for human error. These "helper" scripts will overlap with the auto-style guide correction scripts. Both sets of scripts will reduce the number of mouse clicks that the engineer must do, thereby saving time. Additionally, there will be an automatic style guide checker. This will relieve the model review team of the time consuming and error prone step of doing this manually. Thus, this will result in a more efficient overall process as the model review time will decrease and the subsequent steps will not get delayed from a style guide violation that was missed.

The feature validation will largely remain the same as it is today, but there are some significant improvements with rapid prototyping. The new on-target rapid prototyping and rapid prototyping for distributed applications will allow for more up-front engineering to be conducted. This will result in higher quality models that will reduce the number of errors found throughout the rest of the process. Thus, the overall process will be more efficient and result in a higher quality end product. Additional improvements will result as the modeling tool vendors add more features to the modeling tool such as execution times, tolerances, and units. This will prevent the need to use additional tools and having to worry about translation errors between tools.

One additional feature validation step that will also serve as a requirement validation step is the introduction of formal methods. Formal methods can check for more global conditions that are easy to miss with manual inspections. For example, one part of an algorithm restricts a variable's value to a range of 1 to 10. This variable may be influenced by 9 other variables and 5 calibration settings. A physically remote part of the model my have a condition that checks if the variable is > 12. This can never occur with the current input ranges and calibration settings and is thus dead code. If there is no combination of input ranges and calibration values that would allow the variable to be greater than 12, then this is an unnecessary requirement. A number of other "built-in" checks such as this reachability check are possible. In addition, formal methods tools allow the engineer to ask questions of the model. For example, can cruise control ever be requesting an increase in engine torque at the same time that the ABS is requesting a decrease in engine torque? This feature will allow the engineer to test problems that occurred in the past, were brought up during an FMEA, or that seem like a potential problem. With the proper set of built-in checks and user supplied questions, formal methods will guarantee that the investigated checks will never occur with the given model, input ranges, and other input constraints that were supplied. While the tools to do this are still in development, once formal methods analysis becomes available, the model will be free of "fundamental" flaws and therefore of very high quality. Note, formal methods does not guarantee that the original requirements were met, but only that the model itself is fundamentally sound. While demonstrating that the asked questions satisfy all the stated requirements is a manual effort, formal methods could be used to prove that the requirements have been satisfied assuming that the asked questions are sufficient.

Once the above new process has been fully implemented the model reviews will become significantly easier. The model style guide checking tool will eliminate the need to check modeling style while the formal methods will improve the confidence that the requirements are satisfied. The primary task of the review board will be to check the set of questions that were asked of the model and verify that no requirement was missed.

Model documentation is the same as today, but instead of a manual process to combine the model and other documentation sources into the final format, a tool will do this automatically. Because this is pushbutton easy, there should be no reason for the documentation to be output of date. A web based documentation tool will allow geographically dispersed team members, for example the calibrator who is at the proving grounds, the dyno test engineer who is in the basement, and the software coder who is on the other floor, to have nearly instant access to the latest documentation. This will improve overall process efficiency as there will be less confusion from out of date documentation and the time to wait for documentation to be delivered.

With the ROM and RAM efficiency improvements in the automatically generated code and the increased flexibility of the code generation tools, automatic code generation for a production process is feasible today. There are still some needed automation scripts, such as concerting from the central repository to the autocode tool format, but with these scripts the time consuming and error prone step of generating code can be automated. This will prevent a number of translation bugs from proceeding to subsequent process steps, while also reducing the time for code generation from hours or even days to a matter of seconds. Not only will this be useful for the production code, but can be combined with the on-target rapid prototyping to elevate this one more step from just close to production code to the production code. Autocode will also remove the need for unit testing once the autocode tool is trusted. Lastly, autocode will remove the need for code inspections, as the autocode tool will always be coding style guide compliant and the logic will have been verified previously. Thus, after the initial cost and time of making the automatic code generation pushbutton, autocode will reduce the code generation step, code inspection step, and subsequent testing while improving the quality of the final product.

Low-level drivers and RTOS/Schedulers probably do not provide a competitive advantage, despite the objections of the low-level driver and RTOS/Scheduler coders. If, for example, the in-house RTOS was twice as efficient in ROM, RAM, and CPU usage than the nearest competitor, then that would be a competitive advantage. However, there are commercially available RTOSs (for example see LiveDevices) that are very efficient. Thus, the only competitive advantage would be over companies that do not take the time to find the efficient commercially available RTOSs. Thus, outsourcing the low-level drivers and RTOS/Scheduler code development will allow the engineers to concentrate on intellectual property development which will differentiate the final product from the competition, thereby increasing market share.

The new architecture tool described above will allow individually developed features to be automatically combined into a larger model. This larger model will allow PC-based integration testing allowing earlier detection of integration errors and therefore a lower overall cost to fixing them. If the requirements capture and refinement and all the other steps are successfully implemented, there really should not be any integration errors remaining. Once the process is mature enough and metrics are collected to demonstrate that no integration errors are being detected, this process step can be removed.

With the new architecture tools and appropriate configuration with tracebility links, the build engineer should have an easy time collecting compatible features. This collection should be automated. The build engineer would be the logical person to enter the low-level drivers and the RTOS/Scheduler code from the vendor into the architecture tool so that the build process from this point on is automated.

With all the improved process steps and tools, the in-vehicle/dyno testing should almost become unneeded. The in-vehicle testing should be finding close to zero coding or requirement errors. When the above process becomes mature, the in-vehicle/dyno testing will be primarily concerned with fine-tuning the vehicle performance. A long-term goal is to be able to do all the calibration during the design phases, but it is unlikely that this will be happening in the short to mid term.

With the use of autocode and the elimination of the unit testing, a code coverage tool will no longer be needed. The model coverage tool is still a good metric tool to measure the degree of model coverage that the testing has achieved. This will remove ambiguity especially while the process is maturing. With a mature and trusted process, it may be possible to stop using the model coverage tool as well.

As in today's process, configuration management will be needed. However, with the move from C-based process to a model-based process, the previous file level of version control will not be sufficient. The new models can be quite large, and the engineers (and the entire process) will benefit from a finer-grain control of what gets version controlled. The use of libraries can be done today to address this issue, but this is not necessarily the best solution. Alternative solutions are being investigated by companies like Emmeskey.

5. Summary of Impact Analysis for Desired Process

Process Step/Technology	Cost & Effort to Implement	Expected Benefit	"Perfection"
Configuration Management	Med	Med	
Requirements Capture	High	Low	*
Requirements Tracebilty and Analysis	High	Low	*
Architecture Design and Analysis	High	Med	*
Schedulability Analysis (during design time)	High	Low	*
Central Repository	Med	High	
Feature Controller Algorithm Development	Med	Low	
Discrete Controller Modeling Style Guide	Low	High	
- Automated Style Guide Checker	Med	Med	
- Automated Style Guide Fixer	Med	Med	
Feature Validation			
- On-Target Rapid Prototyping	Low	High	
- Multi-Processor Rapid Prototyping	Med	Med	
Formal Methods	High	Low	*
Model Review	Low	High	
Model Documentation	Med	High	
Low-Level Drives (outsourced)	Low	Med	
Feature Code Generation	Med	High	
Automatic Unit Test Vectors	Low	High	
Model and Code Coverage	Low	Med	
RTOS/Scheduler Code Generation (outsourced)	Low	Med	
PC-based Feature Code Testing (eliminated)			
Code Review (eliminated)			
PC-based Integration Testing	High	Med	
Build the Application	Med	High	
In-vehicle (or dyno) Testing	Low	High	
Code Coverage (eliminated) & Model Coverage	Low	Med	

For the "Perfection" category, this is relating to the commonly understood principle that the first 80% of the work is the easiest and that the last 20% of the solution is the hardest. Those categories marked with '*' are the ones that are trying to push the process towards perfection. While the 'Expected Benefit' of most of these categories are listed as 'low', this is not to imply that the steps should be skipped. The 'low' benefit is in the number of expected errors that are going to be found. However, the errors that are found will be difficult to find errors that could have significant impacts including recall sort of errors. For the requirements capture and tracebility steps, the 'low' benefits is because the near term benefit of these steps is marginal, but the targeted benefit is more from long term usage of the model in which the original requirements become obsolete.

6. Recommended Steps to do Today to start moving towards the desired process

AUTOMATE, AUTOMATE, AUTOMATE – Start the automation process now! Slowly get the engineers use to using tools and start building the infrastructure and glue scripts to make this easy.

6.1. Establish Final Process Upfront

Ideally as much of the final process and tools to implement this final process will be established and debugged ahead of time. Unfortunately, most companies do not have the resources to set aside a group to do this, nor do they want to wait that long to start using good new technology that is available today. Thus, in reality, bits and pieces of the new process will be implemented before other pieces. This will cause some rework to be needed, but hopefully will help in both the short term as well as to help reach the final goal.

A "big picture" goal should be established and shared with both management and the engineers. The engineers should not be inundated with the details, but at least be aware of the final goal which should hopefully explain the changes that they are seeing.

The transition to this new process will most likely be painful and difficult, but is a necessary step. Once properly planned and started, this transition should not be abandoned if it takes a little longer than expected or if unforeseen difficulties arise. If no intermediate milestones are being met, then the migration plans either have to be modified, or a completely new final process needs to be established.

6.2. Update the Software Development Process

The software development process has to allow for change in the process. A big obstacle in many companies is that the engineers and the managers are unwilling to modify the process. A corporate culture has to be established that allows process change and even encourages it (provided that the change is demonstrated to be a good change, changing because one engineer or manager wants to without proof of the benefit of the change is not good). One option to help with this is to establish a formal process-change process. For example, start collecting metrics for today's process that will serve as a baseline to compare against future proposed changes. When a process or tool change is desired, a formal pilot project should be conducted with metrics being collected and compared to the baseline metrics. If the new metrics indicate a net corporate improvement, then the new process or tool can be added to the main line process. To encourage change, one option is to solicit input from the working engineers to find out what their hardest and most time consuming tasks are, and if they would recommend specific tool or process changes. If the engineers are worried about repercussions from management or fellow engineers, an anonymous web page could be used. As these individual items are collected, a summary of these could be placed on a web page in which everyone could go and rate each one, thereby providing a section or division wide opinion instead of just the opinion of one individual. Another option is to give an extra '+' which counts towards their next review to anyone that recommends a process or tool change that gets into the process and does indeed improve the process.

What needs to be done today is to slowly introduce small changes to get managers and engineers use to change. This should make them more receptive to the forthcoming "big" changes.

6.3. Enforce the Software Development Process

Once a process has been established, make sure that the entire division follows it without skipping steps; exceptions should be extremely rare.

Also, management has to allow the engineers the correct amount of time to do all the mandated steps.

6.4. Establish a Matlab Tool's Group

This tools group that will create and maintain the "glue" automation scripts to make a high quality, easy to use, robust software development environment. This group should service many other groups so as to promote reuse of the automation scripts and thereby make it more cost effective. (During performance review time, a '+' could be given for sharing a script with another group, another '+' for a large script that is reused, another '+' for sharing with a group that don't naturally have a connection with, also a '+' when effectively using a script which came from another group.)

The tools group will be responsible for ok-ing the use of a new tool or a new version of a tool, and must make sure that the new tool or tool version will work with all the automation scripts.

The goal is that these glue scripts should be minimized whenever possible and incorporated into the vendor's product whenever possible. Unfortunately no one tool vendor has demonstrated capability to do every step in the process. Also, every company has their own data formats, preferred styles, etc., and it is unlikely that the vendor will support all the company's peculiarities, thereby mandating the need for some automation glue scripts.

7. Conclusion

To fully implement a model-based embedded software development process does take time, effort and money, but if done correctly, in the end the benefits are expected to be significant. According to "Removing Requirement Defects and Automating Test" by Mark R. Blackburn, Robert Busser, and Aaron Nauman "Organizations have demonstrated that the approach can be integrated into existing processes to achieve significant cost and schedule savings."

Author Bio

Scott Ranville is an Embedded Software Tools Consultant and has been doing embedded software tool research since 1995. He started at Ford Motor Company. Contact information: scottranville@softwareberet.com, 303-734-8988.