

Requirement Modeling of Pro Trailer Backup Assist™

Nate Rolfes
 Ford Motor Company

ABSTRACT

Driver assistance features are increasingly dependent upon system architectures that distribute and share responsibilities across various function-based ECUs to minimize cost and redundancy while maximizing engineering efficiency. Clear and accurate system requirements are critical to success, and a robust methodology for validating and testing requirements is essential. Distributed systems are highly sensitive to requirement ambiguity and inaccuracy as they are designed on the assumptions of predictable logical behavior of each functional component. Requirement ambiguity drives variance in implementations which results in system incompatibilities. Errors in requirements lead to faulty implementations that fail not just the component test but also hinder the testing of the entire system of components.

This paper shall discuss a Model-Based Design & Systems Engineering methodology which emphasizes modeling system requirements up-front in a unified simulation environment in order to conduct executable tests of the requirements and validate they deliver the design intent. It shall provide an overview of the requirement modeling methodology, and give real application examples from the development of the 2016MY Ford F-150 Pro Trailer Backup Assist™ feature to demonstrate how requirement modeling can be applied to discover and eliminate requirement and implementation defects through virtual testing and model-based design.

CITATION: Rolfes, N., "Requirement Modeling of Pro Trailer Backup Assist™," *SAE Int. J. Passeng. Cars – Electron. Electr. Syst.* 10(1):2017, doi:10.4271/2017-01-0002.

INTRODUCTION

Requirement modeling is an effective methodology for requirement developers to test and check their specifications via simulation prior to release for implementation. When incomplete or erroneous requirements are released, the burden falls downstream to the implementers to fill the gaps and fix the errors, which often results in delays and defects to the implementation. By creating and testing executable models of the requirements, the requirements author ensures the requirements are faithful to the design intention and does not unnecessarily burden the implementer with design decisions. With the extra time and effort invested up-front, the outcome is win-win: an implementer who has less guesswork and more time to focus on implementation quality, and a requirement author who can expect higher quality implementations that are truer to the design intention. The benefits of requirement modeling are discussed in the cited literature [1][2][3][4], which sets forth some methods and examples of using MATLAB/Simulink® to build and test executable specifications. While requirement modeling can be performed in many executable languages, Simulink® is an efficient choice, as it is often the same tool from which model-based code is generated. Extra translation steps can be avoided when the requirement and implementation/code-generation models can be co-developed in the same tool environment.

Carone [4] provides a walk-through example of the methods to build a requirement model from scratch, which is a helpful introduction to requirement modeling concepts. Lee & Friedman [1][2] discuss the effectiveness of using requirement models to ensure requirement

independence from the design and verification paths, and demonstrate how the use of requirements models in conjunction with coverage tools can be efficiently used to auto-generate test vectors to test the product design model and ensure adequate coverage of the design elements. Their work highlights a fundamental benefit of building a requirement model: the inherent properties of the model can serve as a test “oracle” which can be used throughout the software development cycle to check implementation results. Yang, Bauman, & Beydoun[3] provide a detailed overview of requirement modeling including how to address issues in requirement sources such as “mixed” requirement / implementation details, requirement conflicts, and the benefits of using requirement models on parallel tasks to identify logical errors due to race conditions and deadlocks. Their work provides an example case study from which methods for requirement linking and test harness creation can be used to develop a requirements model-centric design and testing process.

Additional sources provide arguments that the primary source of software defects are due to errors in the requirements [2][8][9][10], especially when distributed systems are involved. They demonstrate how the cost of a requirement error is multiplied the further it is allowed to propagate along the development cycle [6][7]. There are significant cost and process efficiency gains to be achieved by eliminating requirement errors up-front. Software rework is estimated to take up to 60-80% of the cost of software development [8]. Margaret Hamilton, a systems engineer working on Apollo 11, analyzed “the errors that came up during testing” [and] found that nearly three quarters of them were interface errors, not conflicts in timing or

priority” [10]. Her solution was to build a “system of systems” to simulate and test “the relationship between the software, the hardware, and the astronaut” [10]. Her Astronaut-in-the-Loop simulations proved invaluable when “Approximately three minutes before Eagle’s touchdown on the moon, the software over rode a command to switch the flight computer’s priority processing to a radar system...due to a faulty written operations script provided to the crew. The action by the software permitted the mission to safely continue.” [11]

In 2007, NASA conducted a study to identify the sources of software defects across its major programs. The NASA Independent Verification & Validation (IV&V) Program assessed 2,044 high severity software defects from 21 NASA projects. They found that nearly 73% of the defects could be attributed to the requirements phase [9]. The figure below reproduces the findings.

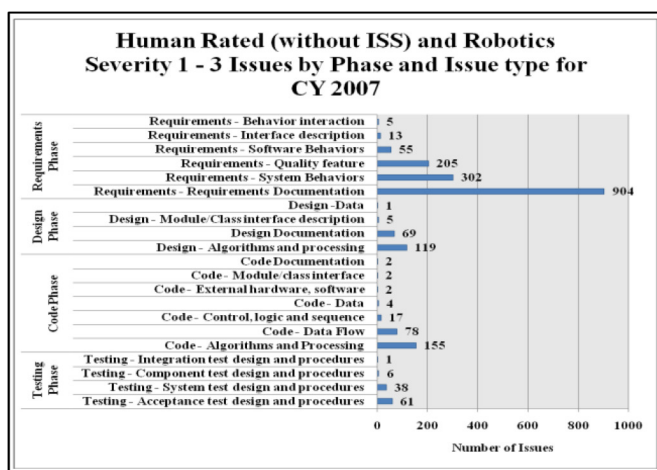


Figure 1. NASA IV&V High Severity Software Defects [9]

Even neglecting the “Requirements Documentation” category, which is a catch-all for missing and incomplete documentation and references (rather than errors in the requirements), we find that the conclusion is unchanged. The requirements phase makes up a majority of the software defects (51%), with over half of these defects due to Requirement System Behaviors (“defects associated with system-level functionality...the ideas of missing, incorrect, incomplete, ambiguous, or unverifiable system requirements / behaviors”).

REQUIREMENT MODELING IN PRACTICE

It is helpful to present a definition for requirement modeling. The author prefers the definition set forth by Lee & Friedman [2]:

“The goal of requirements models is to capture the functional requirement in a clear, concise, analyzable and executable manner, which is typically not possible with natural language. The requirements models can then be used to evaluate the interaction and compatibility of requirements from disparate sources”

The process of translating a document from natural language document to an executable requirement is not always straightforward. Requirements that lack obvious pseudo-code or state machines may require some engineering judgement to decide which, if any, of the

natural language should be included in the executable model. It is helpful to have some best practices to follow. The author has developed a set of guidelines for requirement modeling based on experience:

The Requirement Model Emphasizes States, Signal Interfaces, and Logic

When initially developing a requirement model from a preexisting source, it may be difficult to know which parts are relevant. Start with the state machines, logic, and signal interfaces. Focus on building up the deterministic elements, as these are the basic building blocks to the requirement logic. If an area in the requirements requires extensive additional plant modeling (such as electro-mechanical/thermal/vehicle dynamics), consider simplifying into logical placeholders or breakpoints to represent the event (e.g. a physical voltage might become a state encoded signal of voltage low/normal/high). Software errors due to faulty logic and interfaces can be discovered and fixed quickly without major investment into plant modeling and correlation to real world performance. By simplifying complex natural elements into logical representations, errors in design logic and interfaces can be quickly discovered and eliminated up-front in order to maximize the downstream resources and efforts into real world tuning and plant model correlation.

The Requirement Model Emphasizes Readability, over Efficiency

Consider that the model will go to an engineer or analyst who is not familiar with the specific modeling language or conventions used. Efforts to make the model look as close to an original paper source as possible will help improve the overall readability of the model and make it easier to trace changes between the paper source and the requirements model. Format the model to print easily to paper. Organize the model into subsystems, which contents fit to a computer screen or a printed sheet of paper. These guidelines may conflict with model styling checks for code generation, but this is OK. Requirement models should use formatting constructs that are meant for human readability, not machine code efficiency. The implementer will be better off to receive a readable, printable, and clearly defined requirement model from which they can begin developing the code generation model.

The Requirement Model Is the Requirement, in Executable Form

Translate the requirements directly, even if you think it can be done a better way in the model. Divergence between the specification and the requirement model creates confusion when tracking between model and requirement iterations and makes it difficult to follow-up on error resolution. Do not change the model until you have changed the requirement to match. This helps to ensure a traceable workstream that keeps the requirements and the model aligned. Keep dated and detailed change logs, which correspond to dated comments in the model as you make changes. A seemingly innocuous change to a requirement may have unintended collateral effects on the system that need to be later tested, and you may need to refer back to the change log months after the change was made to understand the reasoning behind the change.

THE DISTRIBUTED SYSTEM V-MODEL

The V-Model is used widely to describe systems engineering and software development processes. A general form of the V-Model is shown below:

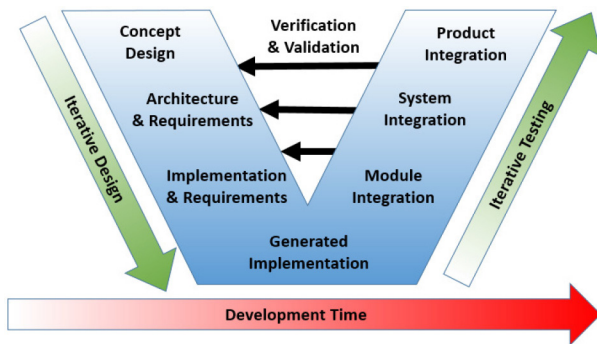


Figure 2. The System & Software Development V-Model

The left side of the V-model represents the process of developing the design of a system from concept to implementation. It involves defining an architecture, developing system and component requirements, and generating an implementation to those requirements. The right side of the V-model represents the process of testing the implementation from a modular, system, and overall product level.

For distributed functions and ECU modules, the V-model can be considered to break apart into separate lanes for each functional module as it develops an implementation to meet the requirements cascaded to it. This concept is shown in the figure below.

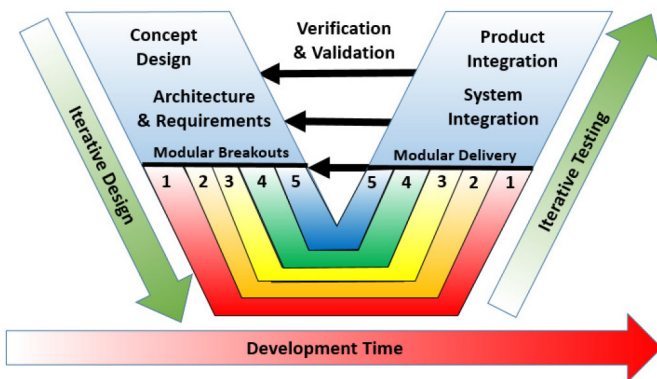


Figure 3. V-Model for Distributed Feature / Function

Systemic issues arise from module breakouts. Each module operates on individual priorities of cost, timing, and capability, making it challenging to achieve a consistent and compatible set of requirements and release dates. Additionally, changes to requirements on one module due to unforeseen implementation constraints may affect requirements for an interfacing component. Ambiguous requirements, which leave some implementation freedom to the module, may allow the implementation to succeed at the module level but fail at the system level. Modules completed early must wait for other modules before full system testing can be completed. Defects in one module can delay further testing of other modules until a workaround is devised or the defect is fixed.

Requirement modeling seeks to minimize these development hardships of the Distributed V-Model by checking the requirement compatibility across modules in the system. In cases when defects are discovered which would prevent further testing of the system, the requirement model can be used in its place. Requirement models can be used to test the impact of a design change to the system, and can be used to find alternative solutions which minimize system impacts. Requirement models can be used to characterize the design intent behavior of the system, in order to generate test cases for each module that can precisely describe the input/output behavior expectations of that module for its role in the distributed system.

SPECIFICATION-MASTER VS. MODEL-MASTER

When establishing a process for requirement modeling, it is important to understand the context of existing requirement-to-implementation processes. The common OEM process is for a master requirement document to be written by component engineers and released to a supplier for implementation. In this case, the requirements document (Specification) is the officially recognized document which all implementation details shall be derived from. The paper shall refer to this process as **Specification-Master**. All implementation issues and discrepancies during testing results will be resolved on the content and understanding of the **Specification-Master** document.

Developing a requirement model from this document will not be fully effective unless the requirement model exactly reflects the content of the document. A workstream is necessary to ensure updates to the requirement model are made for each **Specification-Master** update, and re-testing is performed on the distributed system simulation to ensure no errors have been introduced by the updates. Likewise, errors found in the requirement model testing and simulation process need to be communicated to the specification authors for resolution. Since translation and modeling errors can be introduced in the requirement model that do not exist in the **Specification-Master**, consideration and care should be taken to separate requirement model issues from requirement issues. Strong traceability between the specification and the requirement model is needed so that errors can be quickly identified and resolved, as discrepancies between the model and the Specification-Master will result in false-positives and loss of confidence in the process. The figure below shows an example V-Model for a **Specification-Master** process.

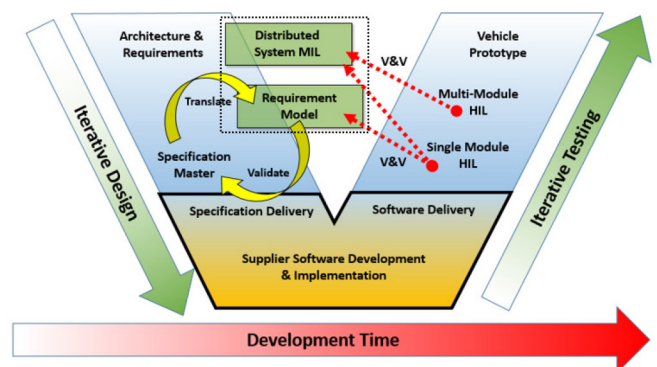


Figure 4. Example Specification-Master Process

A more efficient process for developing requirements to implementation is for the model to be the master requirements document. The paper shall refer to this process as **Model-Master**. In this case, the specification is generated *from* the model. This method minimizes the often tedious and error-prone translation and traceability iteration loop in the **Specification-Master** process. Errors in the **Model-Master** will often prevent the system simulation from continuing until the requirement issue is resolved. A workstream to keep the specification and model aligned is no longer necessary. While more efficient, this process is more difficult to achieve in practice for several reasons. Since there is often a commercial barrier between the requirements and the implementation, it is difficult and/or non-desirable to establish the commercial agreements upon the precise and exact contents of a model's logic, syntax and specificity. Natural language allows for ambiguous and generalizations which make it easier to allow for tolerances and design freedom and protecting intellectual property. Additionally, the implementation details of a requirement may not yet be known enough to provide an executable model, and may even be part of the expected development services and expertise provided by the implementer. Developing a model which is detailed enough for execution and simulation may become counter-productive to the development process. Even when requirements are developed without commercial concerns (i.e. within the same company), there are still obstacles to overcome. If consumers of the model are not trained and primed to read and execute the model, there is a risk for misunderstanding which will create more issues and errors than a **Specification-Master** process. While it is ultimately a more efficient process, the effort and training required to establish a **Model-Master** process that does not contribute more errors than it fixes is not trivial and may not always be the right choice. The figure below shows an example V-Model for a **Model-Master** requirements modeling process.

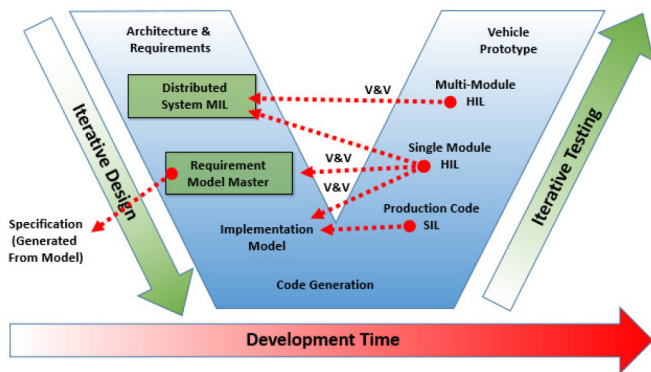


Figure 5. Example Model-Master V-Model Process

PRO TRAILER BACKUP ASSIST™ (PTBA)

Pro Trailer Backup Assist™ (PTBA) is a distributed feature that was introduced on the 2016MY Ford F-150™. The requirement modeling methodology was used throughout the development of the feature, and the findings and results will be used as a case study in this paper. In addition, several supporting simulation devices will be introduced which are used in conjunction with requirements modeling to maximize the effectiveness and efficiency of the process.

The PTBA feature is a distributed system of eight functional modules. The model schematic is shown below:

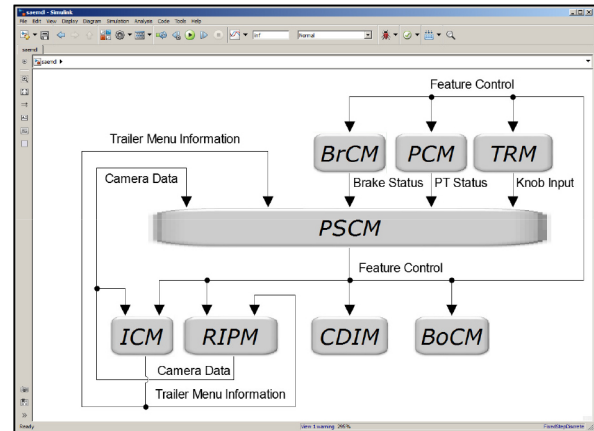


Figure 6. PTBA Model Schematic in Simulink®

The proprietary trailer backing trajectory controller and mode management software are implemented on the Power Steering Control Module (PSCM). Trailer Hitch Angle is sensed by the Rear Image Processing Module (RIPM) and sent to the PSCM module for steering trajectory calculations. The rear camera lighting is managed by the Body Control Module (BoCM). Trailer information management, including selection of trailer and trailer measurement data, is handled by the Instrument Cluster Module (ICM). In addition to storing data, the ICM also is responsible for sounding the informational and warning chimes during feature operation. The ICM and center display infotainment module (CDIM) both provide instructional messages to the driver during feature operation and during the PTBA trailer setup process. The activation button and trailer control knob are managed by the Trailer Lighting Module (TLM). The Braking Control Module (BrCM) provides wheel rotation and odometry information in addition to providing a speed limiting function. This works in conjunction with a speed limiter on the Powertrain Control Module (PCM) to limit vehicle speed and prevent the vehicle from reaching a velocity which exceeds the steering controllability limits and demands of the trailer.

THE DISTRIBUTED FEATURE SIMULATOR (DFS)

The **Distributed Feature Simulator** (DFS) is a collection of Requirement Models-in-the-Loop (MIL) executing in the same environment so that the system interactions and logic amongst them can be tested and validated. The DFS for PTBA will be further described in the sections below.

CAN Communication Requirements

CAN is the primary communication protocol between modules throughout the vehicle. The eight PTBA modules are spread across four independent CAN buses, which are connected via a gateway module. It is useful to build a requirements model of the feature CAN network to validate the CAN signal interfaces to each module and passage through gateway. CAN requirements are exported to test files in what is known as a DBC file, using Vehicle Network Toolbox®.

(VNT), the DBC file can be imported into Simulink® and used to generate the message and signal layout details within the DFS. This architecture is defined in the DBC file and the DFS uses this requirement definition to determine its top-level network layout. Requirements models can then be used to populate inside each functional module and interfaced with the CAN signal I/O as defined in the DBC files. The figure below shows the DFS created for PTBA with the eight modules distributed across four CAN networks, which are connected via a CAN gateway module.

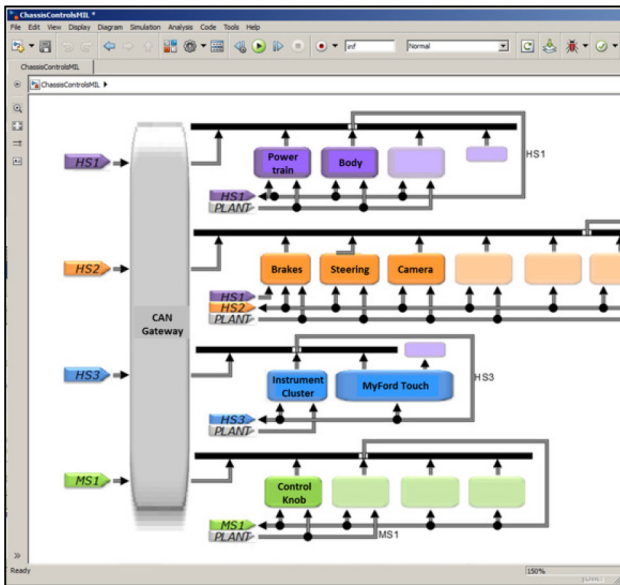


Figure 7. PTBA Distributed Network Simulator

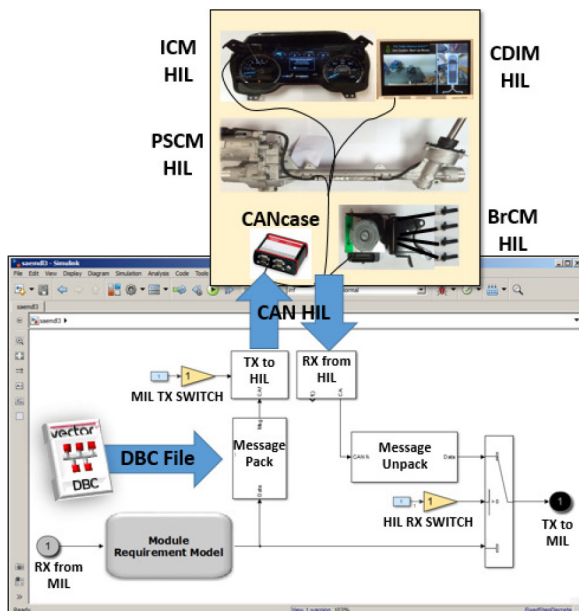


Figure 8. MIL / HIL CAN Switch Schematic using Vehicle Network Toolbox® and Vector® 1640A

The VNT blocks serve a second purpose, not only to import and define the CAN signal interface requirements to the requirements models, but also to directly interface physical hardware modules with

Simulink® for Hardware-in-the-Loop (HIL) testing. This enables both MIL and HIL testing to be conducted concurrently in the same simulation environment and enables the tester to go between the requirement MIL, the implementation MIL, and the physical HIL within the DFS without pausing or stopping the simulation to change configurations. The figure below shows a schematic of the MIL/HIL Switch using VNT and a CAN communication hardware device such as a Vector® 1640A.

HMI and Driver-in-the-Loop Simulation

HMI and Driver-in-the-Loop (DIL) simulation is a powerful tool for “test driving” driver assistance features in order to virtually evaluate the interactions and functionality of the driver inputs to the system. Driver assistance features often require coordinated driver interactions. PTBA requires the driver to interact with an activation button, a 5-way button cluster, instrument cluster instructions and warnings, center stack instructions, warnings, and camera view, a rotary knob, steering wheel, and gear shifter in order to activate or deactivate the feature. DIL introduces a degree of complexity into the system design and testing that must account for driver actions (both expected and unexpected) at every stage of feature operation to ensure that it does not cause faults or unintended behavior. A driving simulator was built and added to the PTBA DFS, which helped to validate the requirements models were able to meet the feature use cases and interactions expected from the driver. Using the MATLAB® GUI Design Editor (GUIDE), representative HMI elements were constructed to simulate the feature HMI and display the requirement model outputs. The figure below shows an example of how a push button on a GUI panel is transmitted to the PTBA DFS modeling environment by setting the value of a gain block in the DFS using the “set_param” command.

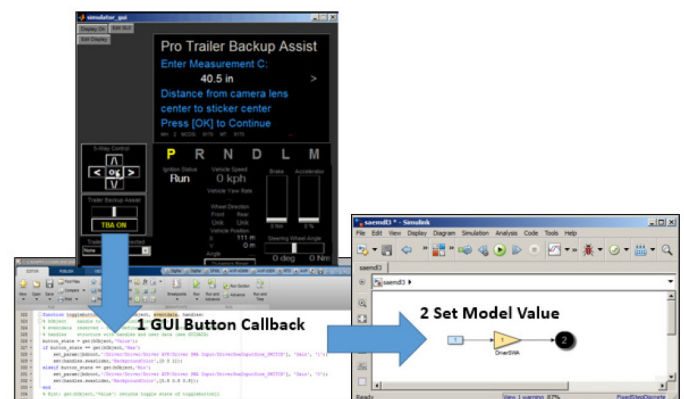


Figure 9. Passing GUI Inputs to the DFS Model

Likewise, GUI displays can also be set based on signal values calculated in the PTBA DFS. Using a MATLAB® function block to call a script during each simulation time step, signal and block values can be queried during simulation using the Run-Time Object API. The figure below shows an example of this where the Steering Wheel Angle is sent to the GUI to be dynamically displayed during simulation in both numerical and slider bar display elements.