# Phase Two Feasibility Study for Software Safety Requirements Analysis Using Model Checking

Gregory Turgeon
PPT Solutions, Inc.
Huntsville, Alabama, USA
greg.turgeon@pptsinc.com

Petra Price
PPT Solutions, Inc.
Huntsville, Alabama, USA
petra.price@pptsinc.com

Keywords: Formal verification, Model checking

**Abstract**

A feasibility study was performed on a representative aerospace system to determine the following: (1) the benefits and limitations to using SCADE®, a commercially available tool for model checking, in comparison to using a proprietary tool that was studied previously [1] and (2) metrics for performing the model checking and for assessing the findings. This study was performed independently of the development task by a group unfamiliar with the system, providing a fresh, external perspective free from development bias.

## 1 Introduction

Reviewing software requirements for a system is an important task, as during the requirements phase approximately 70% of errors are introduced [2]. The earlier those errors are detected, the less costly they are to fix, making the requirements phase an opportune time to find and correct errors. However, typically only 3.5% of errors are removed during the requirements phase [2]. Model checking makes requirements reviews more efficient than manual techniques, as it can identify more defects in less time.

Historically, major drawbacks to the application of formal methods to industry included the reluctance to introduce more time and expense into the software lifecycle process and the fear that formal methods is too proof-heavy and therefore requires expert knowledge of mathematics. However, studies have shown that using a process to formally verify software actually can save time and money during a project [3]. Furthermore, while some knowledge of mathematics is needed, expertise is certainly not necessary because formal methods have evolved from being solely an academic pursuit and is now gaining acceptance in the industrial realm. With this evolution comes improved ease of use and applicability.

A previous feasibility study was performed on a representative aerospace system to assess the viability of a particular set of techniques and tools to perform formal verification of the software requirements [1]. The study explored the types of potential safety issues that could be detected using these tools and determined the level of knowledge required, the appropriateness and the limitations of the tools for real systems, and the labor costs associated with the technique.

The conclusions for the previous study were:

1) The model checking tool set was useful in detecting potential safety issues in the software requirements specification.
2) The particular tool set was not able to model the entire system at one time. The model had to be partitioned, and then assumptions had to be verified about the interfaces between the sections.
3) With basic training in Matlab®/Simulink® and specific training on the tool, engineers were able to become productive with this method in a reasonable time frame.
4) The costs to perform the analysis were commensurate with the system being modeled.

The purpose of this, the subsequent study, was to:

1) Determine if another tool set offers increased scalability and is able to model and verify the entire system with no partitioning.
2) Determine if another tool set is more user-friendly, including better commercially available training, fewer bugs and crashes, and increased ease of use.
3) Develop metrics for the number of requirements that can be modeled and verified per labor hour and the number of safety and other defects found per requirement.
4) Prototype methods for modeling and verifying design aspects.
5) Document training requirements and guidelines.

## 2 Overview of System

In this phase of the study, the same aircraft vehicle management system (VMS) was analyzed as in the previous phase. The VMS determines the overall health status of the aircraft to allow for maximum availability of critical systems. The VMS consists of two redundant computers that are both safety- and mission-critical. The VMS interfaces with the aircraft avionics and the radio communication systems. A data link between the redundant computers provides channel state data and synchronization. The VMS is subdivided into 18 individual managers such as a Communications Manager and Electrical Systems Manager. Due to the proprietary nature of the system, Figure 1 is purposefully general. Elements are encapsulations of functionality either described in the SRS or included by the modelers for clarity and organization.

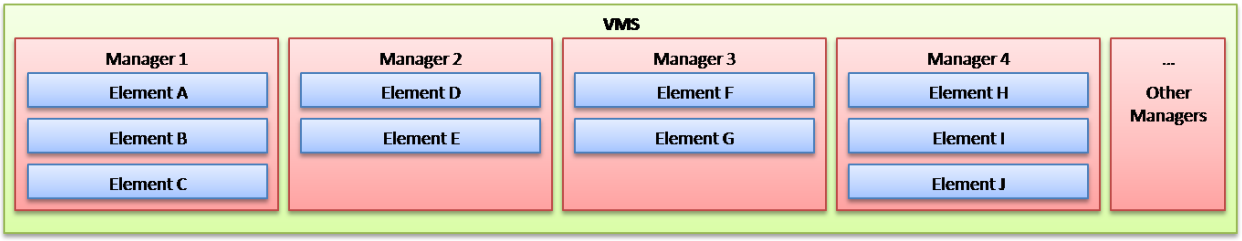


Figure 1: Overview of the System

The VMS performs vehicle mode, flight planning, navigation, and vehicle health functions. There are redundant VMS computers, with one always acting as a master. If the master is determined to be bad, then the slave VMS will assume control of the vehicle. The VMS interfaces with sensors, avionics, and actuators that perform the low-level communications, position, and control surface functions.

The model consisted of 104 block diagrams and 11 state machines, all organized hierarchically. Some state machines were depicted in the requirements document, while the modelers added others when the requirements nicely fit into one.

## 3 Methodology

### 3.1 Stages of Analysis

Analysis begins with a system safety engineer generating system safety requirements based on the system requirements definition and system design. These system safety requirements form the basis of safety properties, which are translated into SCADE® design verification operators. Concurrently, a model of the software requirements is built in SCADE®. Then the SCADE® Design Verifier™ tool is used to determine if the safety properties hold over the entire software requirements model.

For example, the VMS can have one of two designations: master or slave. Having exactly one VMS designated as master is very important for the correct functioning of the entire system, and therefore having one and only one VMS declared as master was a safety property. To determine if this safety property always held, we ran verifiers such as the following: both VMSs are simultaneously master, both VMSs are simultaneously slave, and one VMS is master while the other VMS is slave. All verifiers returned true, meaning that it is possible for the system to have one master, two masters, or no masters; the latter two situations can lead to hazardous outcomes.

Additional algorithm detail is then modeled in Stage 2 using the software design. The safety properties are modified as necessary and formally verified against the design model.

After performing Stages 1 and 2 of the safety analysis, the analyst prepares a safety property report, which lists assumptions made while verifying the system, requirements and design defects found, potential safety hazards, and any other pertinent findings. Based on the safety property report, refinements to the system and software requirements and designs can be made. Changes in requirements or design trigger a new round of analysis. The cycle can continue until no more defects are detected.

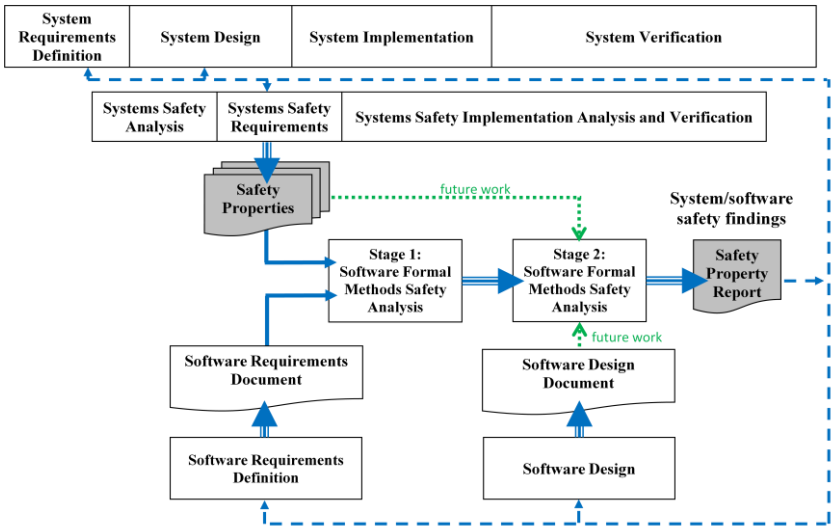


Figure 2: Safety Analysis Stages

### 3.2 Toolset

The tool set consisted of SCADE® version 6.1 including SCADE® Simulator and Design Verifier™. After creating the requirements model in SCADE® Simulator, the model is automatically loaded into the Design Verifier™. The safety properties are also automatically loaded into Design Verifier™. From there, Design Verifier™ determines the status of each property and returns either valid or invalid. Those findings are summarized in the safety property report.

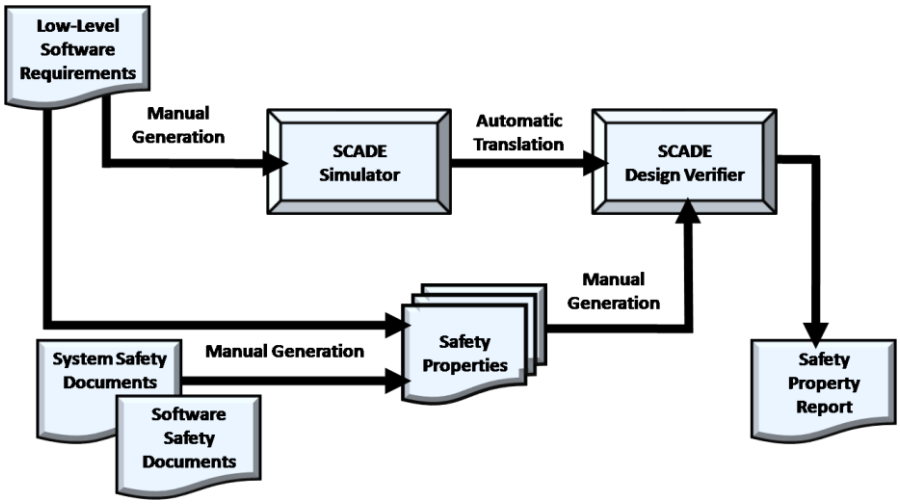


Figure 3: Safety Analysis Toolset

### 3.3 Detailed Methodology

The first step is to create a SCADE® model based on a set of requirements. When creating the model, it is important to keep the model as true to the requirements as possible. Any differences with the requirements from either introducing or eliminating defects weaken the overall effectiveness of the analysis. Any assumptions made in creating the model should be recorded in the safety property report. These requirements assumptions must then be dispositioned by the software requirements engineering team.

After the model is made, the properties to be checked are modeled in SCADE® alongside the requirements model. Four main categories of properties are considered: those based on critical safety conditions, those based on inputs, those based on requirements, and those based on system safety.

The properties based on safety conditions are modeled upon the safety conditions as they are written. Sometimes it is necessary to add constraints to the properties because the safety conditions are vague or unclear. Such constraints are noted and kept to a minimum.

A second method is to model properties based on inputs: each input to the SCADE® model is assessed for its range of values and the possible system repercussions of each of those values. For example, what should occur if both VMSs report a health status of unhealthy? What should occur if a VMS receives an invalid address? These types of properties are written to show that the system behaves as expected based on varying inputs.

A third method is to model properties based on requirements to verify the SCADE® implementation. These properties mainly test functionality, like "If both VMSs are not healthy, then the software in VMS 1 shall assume the responsibility as master." These properties check that the SRS was modeled correctly and behaves as expected. These properties are not very interesting, in that they are meant to show that the modelers faithfully modeled the requirements.

The system safety properties are modeled from a standpoint overlooking the various systems, their overall behavior, and the interactions among them. These properties included ones like "Subsystem A must send signal B to subsystem C before action D takes place."

The next step is to assess the validity of the properties using the SCADE® Design Verifier™, a companion to SCADE® Simulator. The Design Verifier™ is backed by Prover® Plug-In, a commercially available proof engine. The analyst creates a proof objective for each property. He then executes each proof in Design Verifier™, with the results being either true (the property always holds), false (the Verifier™ found a counterexample), or undetermined (meaning either the model has errors or certain settings are preventing the Verifier™ from completing its run).

If a property returns false, then three points of error should be considered. First, the error can come from incorrect implementation of the property. Second, the error can come from incorrect implementation of the model. If both the model and property are correct, then the error can come from an incorrect requirement in the SRS. These sources of error are usually checked in the stated order and the fix to the appropriate area is made, or in the case of an incorrect requirement, the error is documented.

If the error is a result of an incompatible SRS requirement, then the property can be edited to include additional constraints or invariants not specified in the software requirements. The addition of these constraints and invariants can lead to a true property, meaning that the original software requirements lacked specificity, were poorly worded, or otherwise were incompatible with the system. It is important to note that when adding constraints to a property, adding the fewest number of constraints leads to the strongest property. That is, too many constraints can narrow the focus of a property to the point it is no longer useful or meaningful.

This process of modeling properties, checking properties, and modifying properties continues. From this process, certain areas of the requirements specification usually emerge as vulnerable. The design and implementation of these at-risk areas should be reconsidered, as they might affect the safety of the system under analysis.

The entire analysis process can be repeated, incorporating more critical software requirements and design components.

### 3.4 Process Recommendations

These actions were found to save time, make the model checking process easier, and solve several bookkeeping issues. The first three actions are SCADE®-specific. A note about operators: In SCADE®, an operator is a computing unit or block with inputs and outputs. Operators can be pre-defined, like a Boolean *not* operator, or they can be user-defined, such as an operator that computes a custom equation or performs a specific function.

- Hierarchically organize the model using packages. (Packages are like folders.)
- Use structures to compose an operator's inputs and outputs. That way, when an operator gains an input, only the structure changes.
- An operator's contents can be spread across multiple layers so each layer is confined to the dimensions of one screen.
- Use comments to identify requirements and safety properties.
- The organization of proof objectives should match the organization of the model. That is, each operator should have its own set of proof objectives.

## 4 Results

### 4.1 Portion of System Modeled

Out of 1124 requirements in the SRS, we modeled 925 requirements, or 82%. We modeled requirements from every system described in the SRS. Of the 20 systems, we modeled 100% of 17 of them and more than 90% of two additional ones. The remaining 18% of requirements either did not add value to the model or were not related to software. The remaining requirements included ones that described hardware-software interaction and design.

### 4.2 Assumptions

We classified assumptions as any educated guess, clarification, or specification we had to make that we felt was missing from the SRS. We limited assumptions to those clarifications that we felt the requirements authors themselves assumed but did not include in the documented requirements specification. Making the assumptions allowed more of the model to function as expected, which let us investigate deeper and uncover more complex defects. Assumptions were needed to either refine a requirement or to disambiguate among multiple interpretations of a requirement. For example, we had to assume a precedence for every transition in the state machines because no precedence was stated. This kind of assumption was common. An example of a more severe assumption would be assuming that the VMSs are communicating in order for some requirements to always be true. We tracked assumptions both in a spreadsheet and as a note in the model itself. SCADE® does have the capability to create assertions, which are assumptions embedded into the model. We did not use assertions.

In order to create the model and thereby properly formalize the SRS, we made 121 assumptions, which is about one assumption per every 7.6 requirements. The following table gives some examples of assumptions and their associated requirements.

| Requirement | Assumption |
|---|---|
| If the VMS is in State S and is the master, it shall send a synchronization message to the other VMS. | The VMSs are communicating. |
| Thirty-five seconds after power up, the VMS shall begin periodic communications with the other VMS. | "Periodic" communications occur on every cycle. |
| When counting the number of times Event E occurs, the software shall not log more than the limit specified in the spreadsheet. | The name of the spreadsheet and where to find it are not indicated. Assumed that the limit was a constant. |
| The software shall read the RT address of its terminal to determine its own identity. | The RT address signal is latched. |

Table 5: Examples of Assumptions

The following example illustrates how formalizing requirements can lead to early defect detection. A requirement like the following, whose combinatorial logic is ambiguous, can be interpreted in more than one way: "When the following is true, the software shall clear variable A: variable B has been below 88% for at least 2 seconds and either variable C is true or variable D is true and variable E is false." It is unclear which combination should be implemented, as both of the following formal interpretations fit the English specification.

$$(B < 0.88 \text{ for } >= 2 \text{ seconds}) \ \& \ (C \mid D) \ \& \ (!E) \quad (1)$$

$$(B < 0.88 \text{ for } >= 2 \text{ seconds}) \ \& \ (C \mid (D \ \& \ !E)) \quad (2)$$

As the expression (2) was the one intended, simply separating the requirements as shown below resolves the ambiguity.

$$(B < 0.88 \text{ for } >= 2 \text{ seconds}) \ \& \ C \quad (3)$$

$$(B < 0.88 \text{ for } >= 2 \text{ seconds}) \ \& \ D \ \& \ !E \quad (4)$$

### 4.3 Defects Detected

Whereas requirements that needed further refinements warranted assumptions because we felt comfortable making a decision about the intention of the requirement, other requirements problems were not as easy to solve. We called these problems "defects" because we did not feel capable of resolving them ourselves, even with assumptions. For example, several inconsistencies among requirements were found and documented. These include duplicated requirements and conflicting requirements.

Out of the 925 requirements we modeled, we found 198 requirements defects, or about one defect per every 4.7 requirements. Fifty-four (27%) of the defects were found through traditional IV&V methods. Sixty-seven (34%) were found while building the model, and 77 (39%) were found using Design Verifier™. Some representative defects are in the following table.

| Method of Detection | Requirement and Defect |
|---|---|
| Manual IV&V | *If in State A and Input I is true, go to State S.*<br>*If in State A and Input I is true, go to State T.*<br>Requirements conflict because states S and T are mutually exclusive. |
| Model Creation | *If the VMS is Master and the other VMS fails, then it shall remain Master.*<br>What if the VMS is Slave and the other VMS fails? |
| Model Checking | *If the Launch Abort Command input is true, then the launch abort sequence should begin.*<br>Modeled property shows a counterexample where receiving a "Launch Abort Command" does not result in the software signaling to abort the launch sequence. |

Table 6: Example Defects

We classified each requirement that returned a counterexample in one of four categories: catastrophic, critical, marginal, or negligible. Analysis of the system through model verification found 62.5% of all potentially catastrophic defects we found.

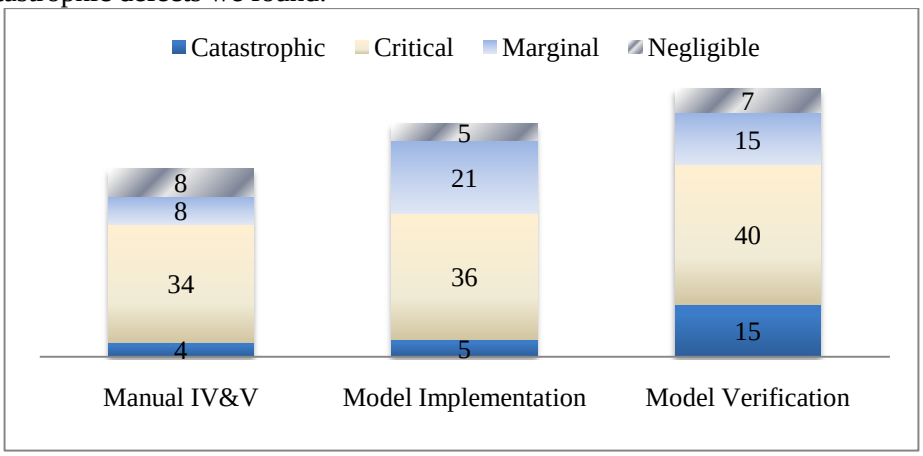


Figure 4: Categorization of Defect Origin

Some counterexamples were helpful in determining the exact reason that a property failed. These counterexamples tended to be short (demonstrated over only a few cycles) and involve few inputs. Other

counterexamples were more complex and difficult to follow. Where possible, we documented a description of the counterexamples as an explanation for a failed property.

We also found problems by verifying some safety properties. The source documents for most of the safety properties were other configured documents specific to the system, such as the Safety Hazard Analysis and Software Safety Analysis. Some safety properties were generated by analyzing the SRS.

We modeled 215 safety properties. Of these 215 properties, 68 or 32% evaluated to false and returned a counterexample. Some examples of safety properties are in the following table.

| Safety Property | Verification Operator Implemented As | Returned |
|---|---|---|
| If the datalink is broken, can both VMSs be master? | After the first cycle, if VMS_A is not communicating and VMS_B is not communicating, then VMS_A is not master or VMS_B is not master. | false |
| An invalid VMS is declared halted. | If a VMS is not valid, then it is in the halted state. | false |
| If the engine oil temperature is out of range a fault will be set. | If the oil temp is less than -50 or greater than 110, then oilTempFault will be true. | false |
| Before the engine started message is sent, the engine start request message must be received. | Is it true that the signal engineStartComAccepted has never been true and the signal engineRunning is true? | false |

Table 7: Example Safety Properties

### 4.4 Return on Investment

We spent approximately 934 hours modeling and verifying properties. Given that we modeled 925 requirements, our modeling and verification rate was about one requirement per hour. We spent 498 hours building the model, 436 hours using Design Verifier™, and 63 hours in training.

Training consisted of three days of formal instruction by representatives of Esterel Technologies. During training, we learned the basic functionalities of SCADE Suite® and practiced making operators. SCADE® is not difficult to learn, especially if one has knowledge of another graphic modeling tool like Simulink®.

Of the 498 hours spent building the model, the initial model creation took 424 hours. There were another 74 hours of editing the model after running Design Verifier™ to fix errors we had introduced to the model and to improve the model's organization.

The time spent in Design Verifier™ accounts for creating the safety properties, running the Verifier™, and editing the properties as needed.

This method is easy to learn and integrates well into the typical software lifecycle. In fact, by using well established rates of productivity based on the number of lines of code [4], we calculated that reviewers would spend approximately 40% less time in requirements review using this method over traditional IV&V methods. We also calculated a total project cost savings of 5% for the system we studied. On top of that savings, the model can be reused to automatically generate code and tests. The time required for training is minimal, though additional maturation with the tool occurs as the modelers gain experience. Considering a post-learning execution time and the number and significance of our findings, this process is not only a feasible inclusion to the software development lifecycle, but is also a valuable asset to the

lifecycle. It can lead to cost savings plus the detection of safety-critical defects that may prevent a catastrophic failure during flight test or operation.

# 5 Conclusions

## 5.1 Training Recommendations

The training provided by Esterel was adequate to begin creating the model. As with most tools, as we gained experience with SCADE®, we found certain methods and shortcuts for improved efficiency. Perhaps a follow-up training session a few weeks after the initial training would have been useful for instruction in some finer details and higher-level usage of the tool's more advanced capabilities. The graphical modeling environment of SCADE® is very similar to that of Simulink®. Engineers familiar with Simulink® should have a smooth transition to SCADE®.

The specific concepts that should be understood prior to training include knowledge of logical operators (and, or, not, implies), Boolean algebra, and state transition diagrams. In general, an engineer who has had college-level math or logic is capable of benefitting from the training.

## 5.2 Objectives Revisited

We had five objectives for this phase of the study. Our conclusions based on those objectives are as follows:

1) *Determine if another tool set is able to model and verify the entire system at one time.* We were able to model 82% of a software requirements document in one SCADE® model. Scalability was not an issue.
2) *Determine if another tool set is more user-friendly, including better commercially available training, fewer bugs and crashes, and increased ease of use.* We found SCADE® to be easy to learn, and the training provided was adequate to begin modeling. However, the first release of SCADE® that we were given was unreliable. We had to back up our work several times a day to prevent losing it because SCADE® crashed often. It was not unusual for SCADE® to crash once or even twice a day for every engineer using it. The second release of SCADE® that we were given was much more reliable and the number of times that this version crashed was few.
3) *Develop metrics for the number of requirements that can be modeled and verified per labor hour and the number of safety and other defects found per requirement.* We modeled approximately one requirement every hour. There was approximately one defect for every 4.7 requirements and one assumption per every 7.6 requirements.
4) *Prototype methods for modeling and verifying design aspects*. We were not able to complete this objective.
5) *Document training requirements and guidelines.* We were not able to complete this objective.

## 5.3 Limitations

There are two main constraints in using this method. One is the faithfulness of the model to the requirements. Is the model behaving as the software behaves? Is the model introducing or eliminating bugs? As the number and complexity of requirements increases, more assumptions are introduced to the model. This limitation applies to any model-checking method, not just SCADE's method. The other limitation is the difficulty of merging work among multiple modelers. We were not able to introduce a satisfactory method of combining the work of multiple modelers. The best we did was to divide the tasks each modeler worked on and manually combine models once a week. The difficultly lies in merging operators that more than one modeler has edited and in connecting and adding inputs and outputs.

SCADE® does not have the capability to "drop in" different versions of operators, even if their inputs and outputs are the same.

### 5.4 Efficacy of Model Checking Methods

This technique has value as a method for checking models. It can be used to determine software functions that contribute to potentially unsafe conditions. It is also a cost-effective means to ensure safety, as it can identify potential software safety defects early in the software's lifecycle, thereby reducing cost. Additionally, this technique can indicate critical software functions that need further testing. This technique also identifies requirements ambiguities. Clarifying these ambiguities helps improve safety and reduces software development and testing costs by minimizing re-work.

We found a total of 198 requirements defects during our analysis, and only 54 of those were found through traditional IV&V methods. The additional 144 defects were discovered while building the model and while running Design Verifier™. Thus traditional IV&V methods missed 73% of the total defects we found.

### 5.5 Recommendations for the Future

The main deficit we recognize is the need for a systematic way to manage and merge versions. We suspect that a free version control tool like CVS would work for managing versions. An efficient way to merge versions proves more elusive.

## References

[1] Greg Turgeon and Petra Price. "Feasibility Study for Software Safety Requirements Analysis Using Formal Methods." Presented at the 27th International System Safety Conference, Huntsville, Alabama, August 2009. Available for purchase at http://www.system-safety.org/products/.

[2] NIST Planning Report 02-3. "The Economic Impacts of Inadequate Infrastructure for Software Testing," May 2002. http://www.nist.gov/director/prog-ofc/report02-3.pdf, last viewed December 2009.

[3] Jonathan P. Bowen and Michael G. Hinchey. "The Ten Commandments of Formal Methods." *Computer*, vol. 28, no. 4, pp. 56-63, April 1995. http://www.is.pku.edu.cn/~qzy/fm/lits/ten_commandmentsFM0.pdf, last viewed December 2009.

[4] Watts S. Humphrey. *Introduction to the Team Software Process*. Addison Wesley, 2000.