A Case for Specification Validation

Mats P.E. Heimdahl

Department of Computer Science and Engineering, University of Minnesota
University of Minnesota Software Engineering Center (UMSEC)

Abstract. As we are moving from a traditional software development process to a new development paradigm where the process it largely driven by tools and automation, new challenges for verification and validation (V&V) emerge. Productivity improvements will in this new paradigm be achieved through reduced emphasis on testing of implementations, increased reliance on automated analysis tools applied in the specification domain, verifiability correct generation of source-code, and verifiably correct compilation. The V&V effort will now be largely focused on assuring that the *formal specifications* are correct and that the *tools* are trustworthy so we can rely on the results of the analysis and code generation without extensive additional testing of the resulting implementation. Most effort has traditionally been devoted to the verification problem. In this position paper we point out the importance of validation and argue that if we fail to adequately address the validation problem problem the impact of verifying code generation and compilation will be limited.

1 Introduction

In software engineering we make a distinction between the *validation* and the *verification* of a software system under development. Verification is concerned with demonstrating that the software implements the functional and non-functional requirements. Verification answers the question *"is this implementation correct with respect to its requirements?"* Validation, on the other hand, is concerned with determining if the functional and non-functional requirements are the right requirements. Validation answers the question *"will this system, if build correctly, be safe and effective for its intended use?"* There is ample evidence that most safety problems can be traced to erroneous and inadequate requirements. Incomplete, inaccurate, ambiguous, and volatile requirements have plagued the software industry since its inception. In a 1987 article [6], Fred Brooks wrote

> *The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements... No other part of the work so cripples the resulting system if done wrong. No other part is as difficult to rectify later.*

We know that the majority of software errors are made during requirements analysis [5, 12, 34, 29], and that requirements errors are more likely to affect the safety of a system than errors introduced during design or implementation [24, 26].

Therefore, to improve the safety of software intensive systems it is critical that the requirements are properly *validated*. Unfortunately, current certification standards, for example, DO-178B [31], as well as the research effort outlined in the Verifiable Software Project focus almost exclusively on various *verification* activities. We find this unfortunate since one of the most critical problems with current certification standards and development practices is a lack of robust and reliable ways of assessing whether the requirements are correct; to gain the full advantage of verifying code generators and compilers we have to develop techniques to determine if the requirements have been adequately validated.

There is a significant effort in the avionics and medical technology industry to reduce the high cost of software development. The current trend is to focus on tools and automation, for example, automatically generating certifiably correct production code from a formal requirements specification or generating MC/DC tests for certification could provide dramatic cost savings. This approach is commonly referred to as model-based development. Since this approach relies heavily on the correctness of the model (or specification) for the correctness of the derived system, our current inability to adequately validate our requirements raises a serious concern regarding the adoption of this type of automation. In the remainder of this position paper we will discuss model-based development in more detail and point to some concerns that must be considered as we adopt verifying translators in software development.

2 Model-Based Development

Traditionally, software development has been largely a manual endeavor. Validation that we are building the right system has been achieved through requirements and specification inspections and reviews. Verification that the system is developed to satisfy its specification is archived through inspections of design artifacts and extensive testing of the implementations (Figure 1). In critical embedded control systems, such as the software controlling aeronautics applications and medical devices, the validation and verification phase (V&V) is particularly costly and consume approximately 50%–70% of the software development resources. Thus, if we could devise techniques to help us reduce the cost of V&V, dramatic cost savings could be achieved. The current trend towards *model-based development* (or specification-centered development [3, 32]) is one attempt to address this problem.

In model-based development, the development effort is centered around a formal description of the proposed software system. For validation and verification purposes, this *formal specification* can then be subjected to various types of analysis, for example, completeness and consistency analysis [17, 19] model checking [15, 9, 10, 20, 11], theorem proving [1, 2], and test case generation [8, 14, 13, 4, 28, 21, 30]. Through manual inspections, formal verification, and simulation and testing we convince ourselves (and any regulatory agencies) that

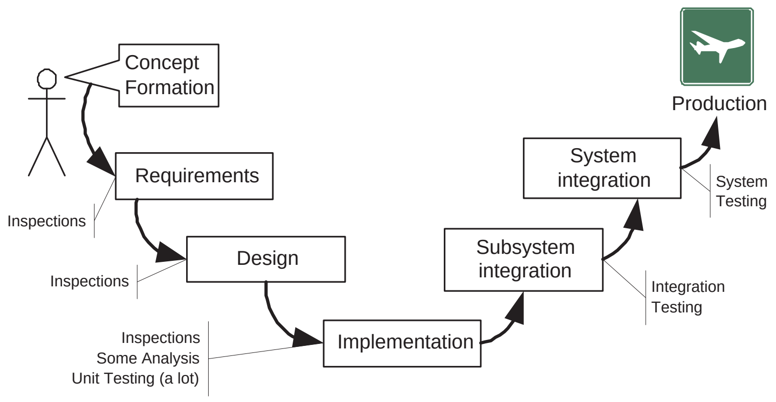

Fig. 1. Traditional Software Development Process.

the software specification possesses desired properties. The implementation is then *automatically and correctly generated* from this specification and little or no additional testing of the implementation is required (Figure 2). There are currently several commercial and research tools that attempt to provide these capabilities—commercial tools are, for example, Esterel and SCADE from Esterel Technologies, Statemate from i-Logix [16], and SpecTRM from Safeware Engineering [25]; and examples of research tool are SCR [18], RSML^{-e} [32], and Ptolemy [23].

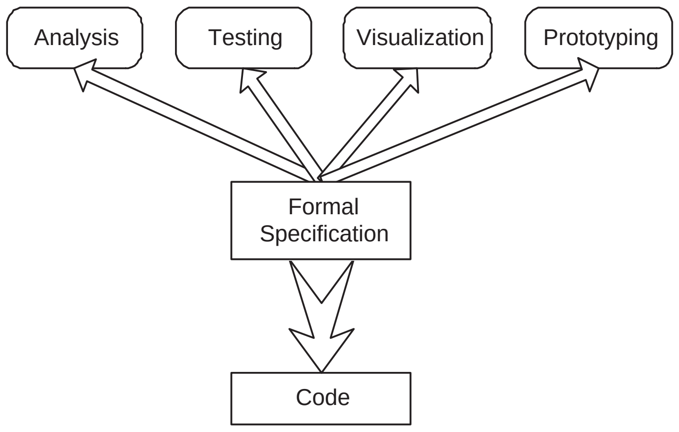

Fig. 2. Specification Centered Development.

The capabilities of model-based development enable us to follow a different process. The development is centered around the formal specification and the V&V has been largely moved from testing and analyzing the code (Figure 1) to analyzing and testing the specification (Figure 3)—the traditional (and, in the

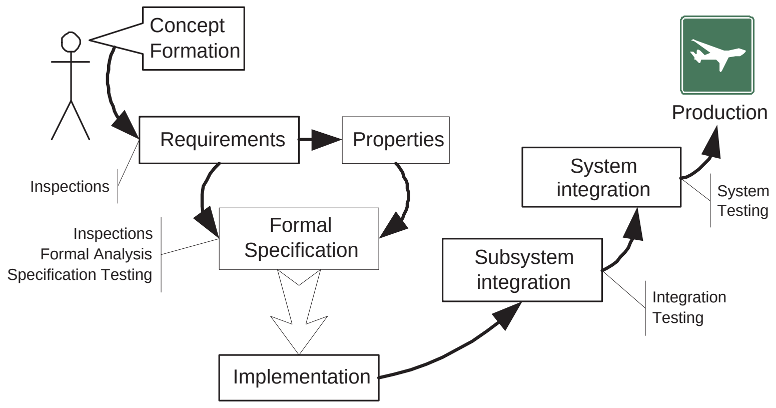

Fig. 3. Specification Centered Development Process.

critical systems domain, very costly) unit testing of code is replaced with testing and analysis of the specification in a hope to provide higher quality at a lower cost.

Note here that, in our opinion, the possibility of reducing or fully automating the costly unit-testing efforts are key to success of specification centered development. We have found very little support for this type of development if modeling and analysis are to be performed in *addition* to what is currently done—these new techniques must either make current efforts more efficient or replace some currently required V&V activity. In either case, our increased reliance on tools requires that they can be trusted—a prime opportunity for verifying translators as well as analysis tools that provide proof justifications and proof explanations. On the other hand, we are now demanding that the formal specification serving as the basis for our development is correct with respect to the customers' true needs; a demand that can only be met through extensive model validation.

In an ongoing project with Rockwell Collins Inc. and NASA Langley Research Center we have investigated model-based development and focused mainly on the verification aspects of the problem. Below we provide a short overview of one of our case examples and discuss some issues that have arisen during the course of this project.

2.1 Overview of a Flight Guidance System

A Flight Guidance System (FGS) is a component of the overall Flight Control System (FCS). It compares the measured state of an aircraft (position, speed, and attitude) to the desired state and generates pitch and roll guidance commands to minimize the difference between the measured and desired state. These guidance commands are both displayed to the pilot as guidance cues on the Primary Flight Display (PFD) and sent to the Autopilot (AP) that moves the control surfaces of the aircraft to achieve commanded pitch and roll.

The internal structure of the FGS can be broken down into the mode logic and the flight control laws. The flight control laws accept information about the aircraft's current and desired state and compute the pitch and roll guidance commands. The mode logic determines which lateral and vertical modes are armed (attempting to lock on to a navigation source) and active (providing guidance to the aircraft) at any given time.

The overall FGS system consists of two identical subsystems, one associated with the left side of the aircraft and one with the right side. In most modes of operation, only one side is active and responds to pilot inputs and produces outputs. The inactive side simply copies its internal state from the active side, serving as a hot backup. In a few critical modes such as Approach and Go Around, both sides of the FGS are active and generate outputs that are compared before they are used.

We have used the mode logic of a FGS as an example in several previous studies [7, 22, 33]. It is an excellent example because it is complex and representative of a class of problems frequently encountered in the design of embedded control systems.

2.2 Modeling Process

In the project, we collected the system requirements as informal "shall" statements. These requirements were relatively mature and well-understood. The next phase, modeling, consisted of constructing by hand an executable model that we believed exhibited the behavior informally stated in the shall statements; in this case we used the RSML^{-e} notation developed at the University of Minnesota. Throughout creation of the model, we continually used the simulation capabilities of the RSML^{-e} execution and analysis environment Nimbus to execute the model and informally confirm that it behaved as we expected. As we built the model, we discovered and corrected numerous ambiguous, unclear, and inconsistent informal requirements.

In the formal verification phase, we manually translated the shall statements into formal properties stated over the model in CTL and merged these formal properties with the translation of the RSML^{-e} model into NuSMV created using a translator developed by the University of Minnesota. Again, the formalization process helped us improve the informal requirements. The NuSMV model checker was then used to confirm whether the property held over the model or not.

When completed, the model of the FGS mode logic consisted of 41 input variables (Boolean and enumerated), 16 small, tightly synchronized hierarchical finite state machines, 122 macro or function definitions, 29 output variables (Boolean and enumerated), and was roughly 160 pages long in its typeset version. We developed 300+ CTL properties based on the informal requirements. A detailed description of the model and its simulation environment is available in [27].

3 The Specification Will be Wrong

As mentioned above, the process of creating a model from the English prose requirements caused us to go back and clarify the English statement of the requirements. In the same way, translating the English statements into SMV also prompted us to go back and clarify the English statement. In addition, the verification that the model satisfied the requirements (formalized as CTL properties) led to some insight into the validation problem. For example, consider the well-validated and non-controversial requirement below.

> *If Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the Flight Control Panel.*

After formalization into CTL, this property did not verify in our model. Using the model-checker we discovered two ways in which this property will not be true. First, if another event arrived at the same time as the HDG switch was pressed, that event could preempt the HDG switch event. Second, if this side of the FGS was not active, the HDG switch event was completely ignored by this side of the FGS. These were two scenarios that were correctly handled in the implementation of the FGS systems, but not captured in any specification. The counterexamples from NuSMV led us to modify the requirement to state

> *If this side is active and Heading Select mode is not selected, Heading Select mode shall be selected when the HDG switch is pressed on the FCP (providing no higher priority event occurs at the same time).*

While longer and more difficult to read than the original statement, it has the advantage of being a more accurate description of the system's behavior. Of course, we also had to clearly define what a "higher priority"' event was.

We found that the process of proving the properties forced us to go back and modify virtually all of our original English requirements; consequently, all formal specification properties also had to be modified.

It also became clear to the engineers formalizing the properties that great care needs to be taken when formulating SMV properties to ensure that their proofs are meaningful. For example, in the modelling of the FGS we frequently used macros to encapsulate commonly used properties, for example, we might encapsulate a complex condition in a macro named "When_Lateral_Mode_Manually_Selected". The macros were frequently used when the properties were stated as SMV properties. In most cases, the macro was used as the antecedent of an implication, for example,

```
AG(m_When_Lateral_Mode_Manually_Selected.result ->
   Onside_FD_On)
```

Naturally, if the macro "When_Lateral_Mode_Manually_Selected" was over-constrained in the model (or even contradictory and thus always false) this proof would succeed but it would be rather meaningless.

To summarize, when developing formal models of any substantial system, the models will most likely be incorrect with respect to the real needs of the system. In our case, the three complementary models—informal English language requirements, requirements formalized as CTL properties, and an executable formal model—served to check each other in a rigorous validation process. Had we only built the executable model and validated it through testing, chances are significant flaws would have remained. Similarly, had we been blessed with a correct-by-construction tool that would have helped us refine our 300+ CTL properties to an implementation, the implementation would certainly have been grossly incorrect with respect to the customers' real needs. It is clear that a *rigorous validation process* must be in place to ensure that any formal artifacts serving as the basis for downstream automation are correct; without this validation any breakthroughs in verifiably correct code generation and compilation will achieve limited success.

4 Loss of "Collateral Validation"

The goal of adopting model-based development is to reduce the high cost of software development. The hope is that by relying on tools and automation, for example, automatically generating certifiably correct production code from a formal requirements specification or generating MC/DC tests for certification, we could provide dramatic cost savings. These cost savings will be achieved by replacing time consuming and costly manual processes, for example, design, coding, and definition of test-cases, with tools. As mentioned above, our current inability to adequately validate our requirements raises a serious concern regarding the adoption of this type of automation. Manual processes, may that be design, coding, testing, or putting a medical device through clinical studies, draw on the collective experience and vigilance of many dedicated software professionals; professionals that provide *"collateral validation"* as they are working on the software. Experienced professionals designing, developing code, or defining test cases provide additional validation of the software system; if there is a problem with the specified functionality of the system, they have a chance of noticing and taking corrective action. As an example, consider the requirements example from the previous section. Although the facts that the FGS had to be active and that no higher-priority events were received at the same time were not explicitly sated in the requirements, the engineers implemented the FGS functionality correctly; these problems were caught in the manual development process. When replacing these manual efforts with automation, proper validation of the formal requirements specifications on which the automation is based becomes absolutely essential; there may be no safeguards in the downstream development activities to catch critical flaws in the formal model—the collateral validation is lost.

Naturally, the tools we use in the validation process may lead to additional problems. For example,

1. If the specification execution environment misrepresents the semantics of the specification language, all testing and validation done in the specification domain is invalid.

2. If the code generation is incorrect, the resulting implementation will naturally be wrong (and it may not be caught since we are now reducing testing in the code domain).
3. If any of our analysis tools applied in the specification domain provide false negatives (they fail to catch a faulty specification), we may mistakenly accept a specification as correct and use it for code generation (again, this problem is unlikely to be caught with the reduced code testing).

Solutions to such problems must be provided before these promising techniques can be effectively used in the development of critical systems. The research agenda laid out in this workshop promises to address some of the concerns related to the tools used in model-based development, for example, verifying translators and trusted proof checkers would address issues 2 and 3 above. Unfortunately, execution environments, code generators, and analysis tools are not simple pieces of software and it is highly unlikely that we will be able to provide the level of confidence necessary to trust a specific tool as a development tool [31] in critical systems development. Also, consider tool evolution and the cost of reverification of evolving tools and the situation looks grim.

5 No Need For Perfection

As we ponder how new analysis techniques and development tools can be effectively deployed in the critical systems domain, we cannot loose track of one important fact—*although perfection and full verification is the goal, perfection is not necessary for deployment and highly effective use.* After all, our aim with increased use of tools is to replace costly, time consuming, and error prone manual tasks such as inspections and testing, and all that is really necessary from our tools is that they are *better* than the manual tasks they replace. Unfortunately, we do not know much about how error prone our manual processes really are, nor do we know how to compare the effectiveness of an automated process to a manual process—much important analytical and empirical research is needed to help answer this question.

6 Summary

The emergence of formal modeling languages acceptable to practicing engineers and the development of powerful analysis tools, for example, model checkers, will enable a new development paradigm relying heavily on the use of automated tools for analysis and code generation—a development paradigm often referred to as model-based development. As a community, we are now in a position to bring the full power of formal software development to fruition. As discussed in this position paper, however, we have to approach this opportunity cautiously. All formal development efforts rely on a correct specification as a basis for development and verification; this puts enormous demands on the validation of the specification—the specification simply must be correct with respect to the customers' needs. Unfortunately, research efforts outlined in the Verifiable Software Project focuse almost exclusively on various *verification* activities. We find this troublesome; to gain the full advantage of verifying code generators and compilers we have to concurrently develop techniques to determine if the specifications have been adequately validated. It would be highly disappointing if the enormous advantages in verification technology we have seen the last decade, and will most likely see in the future, are used to verify that faulty specifications are implemented correctly. A few well publicized failures are enough to make widespread industry adoption and regulatory acceptance very difficult and set our efforts back a decade—let us make sure that this does not happen.

Acknowledgements

The author wishes to acknowledge the contributions of Dr. Steven P. Miller, Dr. Michael W. Whalen, and Dr. Alan Tribble of Rockwell Collins for their modeling and analysis efforts, and valuable discussions; the ongoing support of this work by Ricky Butler, Kelly Hayhurst, and Celeste Bellcastro of the NASA Langley Research Center; and the efforts of his current and former graduate students Anjali Joshi, Ajitha Rajan, Yunja Choi, Sanjai Rayadurgam, George Devaraj, and Dan O'Brien of the University of Minnesota.

References

1. Myla Archer, Constance Heitmeyer, and Steve Sims. TAME: A PVS interface to simplify proofs for automata models. In *User Interfaces for Theorem Provers*, 1998.
2. S. Bensalem, P. Caspi, C. Parent-Vigouroux, and C. Dumas. A methodology for proving control systems with Lustre and PVS. In *Proceedings of the Seventh Working Conference on Dependable Computing for Critical Applications (DCCA 7)*, pages 89–107, San Jose, CA, January 1999. IEEE Computer Society.
3. Valdis Berzins, Luqi, and Amiram Yehudai. Using transformations in specification-based prototyping. *IEEE Transactions on Software Engineering*, 19(5):436–452, May 1993.
4. M. R. Blackburn, R. D. Busser, and J. S. Fontaine. Automatic generation of test vectors for SCR-style specifications. In *Proceedings of the 12th Annual Conference on Computer Assurance, COMPASS'97*, June 1997.
5. B. Boehm. *Software Engineering Economics*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
6. F. Brooks. No silver bullet: : Essence and accidents of software engineering. *IEEE Computer*, pages 10–19, April 1997.
7. R. Butler, S. Miller, J. Potts, and V. Carreno. A formal methods approach to the analysis of mode confusion. In *17st Digital Avionics Systems Conference (DASC'98)*, volume 1, pages C41/1 – C41/8, Belllevue, WA, October 1998.
8. J. Callahan, F. Schneider, and S. Easterbrook. Specification-based testing using model checking. In *Proceedings of the SPIN Workshop*, August 1996.
9. W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1998.

10. Y. Choi and M. Heimdahl. Model checking RSML^{-e} requirements. In *Proceedings of the 7th IEEE/IEICE International Symposium on High Assurance Systems Engineering*, pages 109–118, Tokyo, Japan, October 2002.
11. Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.
12. A. Davis. *Software Requirements: Object, Function, and States.* Prentice-Hall, Englewood Cliffs, NJ, 1993.
13. A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proceedings of TACAS'97, LNCS 1217*, pages 384–398. Springer, 1997.
14. Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. *Software Engineering Notes*, 24(6):146–162, November 1999.
15. O. Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
16. D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtull-Trauring, and M. Trakhtenbrot. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
17. Mats P. E. Heimdahl and Nancy G. Leveson. Completeness and consistency in hierarchical state-base requirements. *IEEE Transactions on Software Engineering*, 22(6):363–377, June 1996.
18. C. Heitmeyer, A. Bull, C. Gasarch, and B. Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the Tenth Annual Conference on Computer Assurance, COMPASS 95*, 1995.
19. C.L. Heitmeyer, R.D. Jeffords, and B.G. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996.
20. Constance Heitmeyer, James Kirby Jr., Bruce Labaw, Myla Archer, and Ramesh Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Transactions on Software Engineering*, 24(11):927–948, November 1998.
21. Robert Jasper, Mike Brennan, Keith Williamson, Bill Currier, and David Zimmerman. Test data generation and feasible path analysis. In *Proc. of Int'l Symp. on Software Testing and Analysis*, pages 95–107, August 1994.
22. Anjali Joshi, Steven P. Miller, and Mats P.E. Heimdahl. Mode confusion analysis of a flight guidance system using formal methods. In *22nd Digital Avionics Systems Conference (DASC'03)*, pages 2.D.1–1 – 2.D.1–11, October 2003.
23. Edward A. Lee. Overview of the ptolemy project. Technical Report Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003.
24. N. Leveson. *Safeware: System Safety and Computer.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.
25. Nancy G. Leveson, Mats P.E. Heimdahl, and Jon Damon Reese. Designing Specification Languages for Process Control Systems: Lessons Learned and Steps to the Future. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, volume 1687 of *LNCS*, pages 127–145, September 1999.
26. R. Lutz. An overview of REFINE 2.0. In *Proceedings of the First ACM SIGSOFT Symposium on the Foundations of Software Engineering*, 1993.

27. S. Miller, A. Tribble, T. Carlson, and E. Danielson. Flight guidance system requirements specification. Technical Report CR-2003-212426, NASA Langley Research Center, June 2003. Available at http://techreports.larc.nasa.gov/ltrs/refer/2003/cr/NASA-2003-cr212426.refer.html.
28. A. Jefferson Offutt, Yiwie Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, October 1999.
29. C. Ramamoorthy, A. Prakesh, W. Tsai, and Y. Usuda. Software engineering: Problems and perspectives. *IEEE Computer*, pages 191–209, October 1984.
30. Sanjai Rayadurgam and Mats P.E. Heimdahl. Coverage based test-case generation using model checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2001)*, pages 83–91. IEEE Computer Society, April 2001.
31. RTCA. *Software Considerations In Airborne Systems and Equipment Certification*. RTCA, 1992.
32. Jeffrey M. Thompson, Mats P.E. Heimdahl, and Steven P. Miller. Specification based prototyping for embedded systems. In *Seventh ACM SIGSOFT Symposium on the Foundations on Software Engineering*, number 1687 in LNCS, pages 163–179, September 1999.
33. A. Tribble and S. Miller. Safety analysis of a flight guidance system. In *21st Digital Avionics Systems Conference (DASC'02)*, volume 2, pages 13C1–1 – 13C1–10, Irvine, CA, October 2002.
34. A. van Schouwen. The A-7 requirements model: Re-examination for real-time systems and an application to monitoring systems. Technical Report 90-276, Queens University, Hamilton, Ontario, 1990.