A Formal Approach to Requirements-Based Programming

Michael G. Hinchey, James L. Rash
NASA Goddard Space Flight Center
Information Systems Division
Greenbelt, Maryland, USA
michael.g.hinchey, james.l.rash@nasa.gov

Christopher A. Rouff
SAIC
Advanced Concepts Business Unit
McLean, VA 22102
rouffc@saic.com

Abstract

No significant general-purpose method is currently available to mechanically transform system requirements into a provably equivalent model. The widespread use of such a method represents a necessary step toward high-dependability system engineering for numerous application domains. Current tools and methods that start with a formal model of a system and mechanically produce a provably equivalent implementation are valuable but not sufficient. The "gap" unfilled by such tools and methods is that the formal models cannot be proven to be equivalent to the requirements. We offer a method for mechanically transforming requirements into a provably equivalent formal model that can be used as the basis for code generation and other transformations. This method is unique in offering full mathematical tractability while using notations and techniques that are well known and well trusted. Finally, we describe further application areas we are investigating for use of the approach.

Key Words: Validation, verification, formal methods, automatic code generation, requirements-based programming

1. Introduction

Development of a system that will have a high level of reliability requires the developer to represent the system as a formal model that can be proven to be correct. Through the use of currently available tools, the model can then be automatically transformed into code with minimal or no human intervention to reduce the chance of inadvertent insertion of errors by developers. Automatically producing the formal model from customer requirements would further reduce the chance of insertion of errors by developers.

The need for ultra-high dependability systems increases continually, along with a correspondingly increasing need to ensure correctness in system development. By "correctness", we mean that the implemented system is equivalent to the requirements, and that this equivalence can be proved mathematically.

Available system development tools and methods that are based on formal models provide neither automated generation of the models from requirements nor automated proof of correctness of the models. Hence, today there is no automated means to produce a system or a procedure that is a provably correct implementation of the customer's requirements. Further, requirements engineering as a discipline has yet to produce an automated, mathematics-based process for requirements validation.

2. Problem Statement

Automatic code generation from requirements has been the ultimate objective of software engineering almost since the advent of high-level programming languages, and calls for a "requirements-based programming" capability have become deafening [9]. Several tools and products exist in the marketplace for automatic code generation from a given model. However, they typically generate code, portions of which are never executed, or portions of which cannot be justified from either the requirements or the model. Moreover, existing tools do not and cannot overcome the fundamental inadequacy of all currently available automated development approaches, which is that they include no means to establish a provable equivalence between the requirements stated at the outset and either the model or the code they generate.

Traditional approaches to automatic code generation, including those embodied in commercial products such as Matlab [20], in system development toolsets such as the B-Toolkit [19] or the VDM++ toolkit [17], or in academic research projects, presuppose the existence of an explicit (formal) model of reality that can be used as the basis for subsequent code generation, as shown in Figure 1 (a). While such an assumption is reasonable, the advantages and disad-

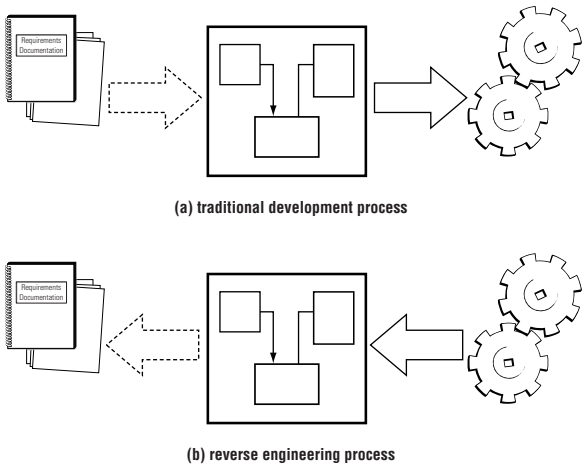

Figure 1. (a) Traditional software development process from requirements to code, and (b) reverse engineering from code to a system description.

vantages of the various modeling approaches used in computing are well known and certain models can serve well to highlight certain issues while suppressing other less relevant details [22]. It is clear that the converse is also true. Certain models of reality, while successfully detailing many of the issues of interest to developers, can fail to capture some important issues, or perhaps even the most important issues. Existing reverse-engineering approaches suffer from a similar plight. Typically (see Figure 1 (b)), a model is extracted from an existing system and is then represented in various ways, for example as a digraph [21]. The re-engineering process then involves using the resulting representation as the basis for code generation, as above.

2.1. Specifications, Models, and Designs

The model on which automatic code generation is based is referred to as a design, or more correctly, a design specification. There is typically a mismatch between the design and the implementation (sometimes termed the "specification-implementation gap"), in that the process of going from a suitable design to an implementation involves many practical decisions that must be made by the automated tool used for code generation without any clear-cut justifications, other than the predetermined implementation decisions of the tool designers. There is a more problematic "gap", termed the "analysis-specification gap," that emphasizes the problem of capturing requirements and adequately representing them in a specification that is clear, concise, and complete. Unless the specification is formal, proof of correctness is impossible [1]. Unfortunately, many are reluctant to embrace formal specification techniques, believing them to be difficult to use and apply [2] [7], despite many industrial success stories [11] [12] [13] [24]. Our experience at NASA Goddard Space Flight Center (GSFC) has been that while engineers are happy to write descriptions as natural language scenarios, or even using semi-formal notations such as Unified Modeling Language (UML) use cases, they are loath to undertake formal specification.

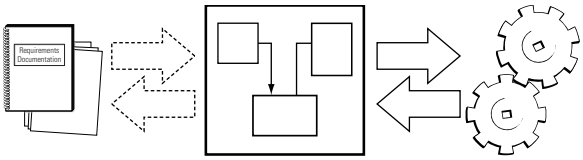

Figure 2. The R2D2C approach, generating a formal model from requirements and producing code from the formal model, with automatic reverse engineering.

2.2. A Novel Approach

The approach described herein, provisionally named R2D2C ("Requirements to Design to Code"), provides mathematically tractable round-trip engineering for system development.

In this approach, engineers (or others) may write requirements as scenarios in constrained (domain-specific) natural language, or in a range of other notations (including UML use cases). These will be used to derive a formal model (Figure 2) that is guaranteed to be equivalent to the requirements stated at the outset, and which will subsequently be used as a basis for code generation. The formal model can be expressed using a variety of formal methods. Currently we are using CSP, Hoare's language of Communicating Sequential Processes [15] [16], which is suitable for various types of analysis and investigation, and as the basis for fully formal implementations as well as automated test case generation, etc.

R2D2C is unique in that it allows for full formal development from the outset, and maintains mathematical soundness through all phases of the development process, from requirements through to automatic code generation. The approach may also be used for reverse engineering, that is, in retrieving models and formal specifications from existing code (Figure 2). The method can also be used to "paraphrase" (in natural language, etc.) formal descriptions of existing systems. In addition, the approach is not limited to

generating executable code. It may also be used to generate business processes and procedures, and we are currently experimenting with using it to generate instructions for robotic devices to be used on the Hubble Robotic Servicing Mission (HRSM). We are also experimenting with using it as a basis for an expert system verification tool, and as a means of capturing expert knowledge for expert systems. Such potential applications will be described in Section 4.

3. Technical Approach

Section 3.1 describes R2D2C at a relatively high level. Section 3.2 describes an intermediate version of the approach for which we have built a prototype tool [23], and with which we have successfully undertaken some examples.

3.1. R2D2C

The R2D2C approach involves a number of phases, which are reflected in the system architecture described in Figure 3. The following describes each of these phases.

D1 Scenarios Capture: Engineers, end users, and others write scenarios describing intended system operation. The input scenarios may be represented in a constrained natural language using a syntax-directed editor, or may be represented in other textual or graphical forms.

D2 Traces Generation: Traces and sequences of atomic events are derived from the scenarios defined in D1.

D3 Model Inference: A formal model, or formal specification, expressed in CSP is inferred by an automatic theorem prover – in this case, ACL2 [18] – using the traces derived in phase 2. A deep[1] embedding of the laws of concurrency [14] in the theorem prover gives it sufficient knowledge of concurrency and of CSP to perform the inference. The embedding will be the topic of a future paper.

D4 Analysis: Based on the formal model, various analyses can be performed, using currently available commercial or public domain tools, and specialized tools that are planned for development. Because of the nature of CSP, the model may be analyzed at different levels of abstraction using a variety of possible implementation environments. This will be the subject of a future paper.

D5 Code Generation: The techniques of automatic code generation from a suitable model are reasonably well understood. The present modeling approach is suitable for the application of existing code generation techniques, whether using a tool specifically developed for the purpose, or existing tools such as FDR [5], or converting to other notations suitable for code generation (e.g., converting CSP to B [3] and then using the code generating capabilities of the B Toolkit).

1 "Deep" in the sense that the embedding is semantic rather than merely syntactic.

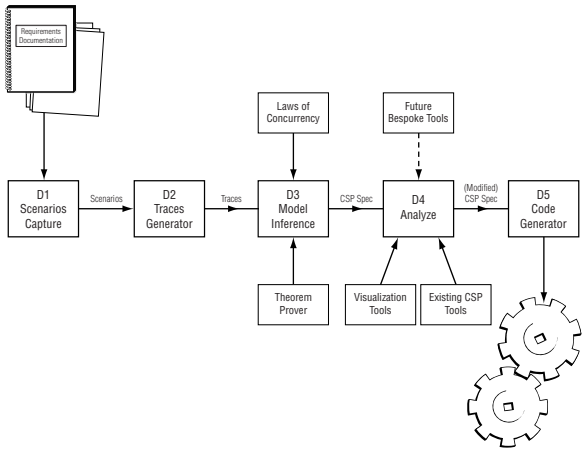

Figure 3. The entire process with D1 thru D5 illustrating the development approach and R1 thru R4 the reverse engineering.

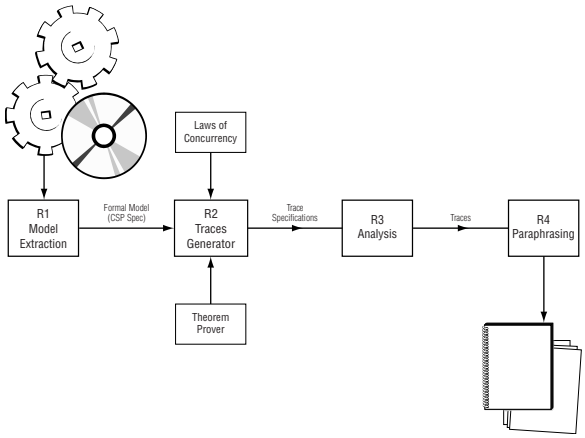

Figure 4. Reverse engineering of system using R2D2C.

It should be re-emphasized that the "code" that is generated may be code in a high-level programming language, low-level instructions for (electro-) mechanical devices, natural-language business procedures and instruc-

tions, or the like. As Figure 4 illustrates, the above process may also be run in reverse:

R1 *Model Extraction*: Using various reverse engineering techniques [25], a formal model expressed in CSP may be extracted.

R2 *Traces Generation*: The theorem prover may be used to automatically generate traces based on the laws of concurrency and the embedded knowledge of CSP.

R3 *Analysis*: Traces may be analyzed and used to check for various conditions, undesirable situations arising, etc.

R4 *Paraphrasing*: A description of the system (or system components) may be retrieved in the desired format (natural language scenarios, UML use cases, etc.).

Paraphrasing, whereby more understandable descriptions (above and beyond existing documentation) of existing systems or system components are extracted, is likely to have useful application in future system maintenance for systems whose original design documents have been lost or systems that have been modified so much that the original design and requirements document do not reflect the current system.

3.2. Short-cut R2D2C

The approach described in Section 3.1 is the way that R2D2C is intended to be applied, from requirements specification through to code generation. However, the approach requires significant computing power in the form of an automated theorem prover performing significant inferences based on traces input and on its "knowledge" of the laws of concurrency. While this is well warranted for certain applications, it is likely to be beyond the resources of many developers and organizations. As a practical concession, we also define a reduced version of R2D2C called the short-cut version (Figure 5), whereby the use of a theorem prover is avoided, yet without sacrificing high confidence in the validity of the approach. The following describes each of the phases for the shortcut R2D2C:

S1 *Scenarios Capture*: As before, intended system behavior is described by scenarios input in natural language or an appropriate graphical or semi-formal notation.

S2 *Translation to Intermediate Notation*: Scenarios are translated to an intermediate notation, termed EzyCSP, which is a simple natural language-like subset of CSP that can be used to describe a large number of situations and scenarios (recall that scenarios are domain specific).

S3 *Analysis*: While far more simple than CSP, EzyCSP allows some simple analyses to be performed.

S4 *Implementation in Java*: EzyCSP is sufficiently simple that it may easily be translated to Java and executed.

This simplified or short-cut approach clearly has significant disadvantages when compared to our full approach. Firstly, the correctness of the development process is contingent on the correctness of both the translation of scenarios to the intermediate (EzyCSP) notation and the translation of EzyCSP to Java. However, the correctness of the translators for these is assured via a proof of correctness undertaken with the ACL2 theorem prover. Secondly, we do not have a reverse process suitable to support reverse and (ultimately) re-engineering, for free. However, a Java-to-EzyCSP translator would certainly be possible for highly constrained subsets of Java.

The significant advantage of this simplified approach, however, is that although a proof of correctness involving a theorem prover is still required, this is required exactly once and would be performed by the support system developers (presumably expert in the art). This is significantly less expensive computationally than using a theorem prover in the development of each individual application.

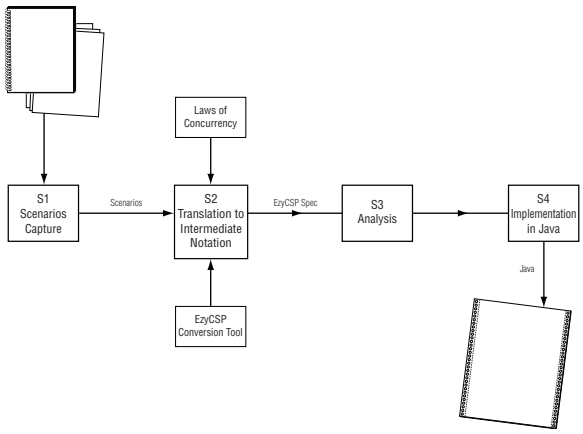

Figure 5. Short cut R2D2C.

4. Application Areas

The motivation for this work was the need for requirements-based programming for ultra high dependability systems. The method described in this paper is applicable in a number of areas, with potentially significant value in the following:

Sensor Networks

NASA is currently conducting research and development on sensor networks for planetary and solar system exploration as well as to support its Mission to Planet Earth.

An example of a sensor network for solar system exploration is the Autonomous Nano Technology Swarm mission (ANTS) [4], which is at the concept development phase. This mission will send 1,000 pico-class (approximately 1 kg) spacecraft to explore the asteroid belt. The ANTS spacecraft will act as a sensor network making observations of asteroids and analyzing their composition. Sensor networks are also being considered for planetary (e.g., Martian) exploration, to yield scientific information on weather and geology. For the Mission to Planet Earth, sensor networks are already being researched and developed towards capabilities for early warnings about natural disasters and climate change.

Projected NASA sensor networks are highly distributed autonomous "systems of systems" that must operate with a high degree of reliability. The solar system and planetary exploration networks will necessarily experience long communications delays with Earth, will partly and occasionally be out of touch with the Earth and mission control for long periods of time, and must operate under extremes of dynamic environmental conditions. Due to the complexity of these systems as well as their distributed and parallel nature, they will have an extremely large state space and will be impossible to test completely using traditional testing techniques. The more "code" or instructions that can be generated automatically from a verifiably correct model, the less likely that human developers will introduce errors. In addition, the higher the level of abstraction that developers can work from, as is afforded through the use of scenarios to describe system behavior, the less likely that a mismatch will occur between requirements and implementation and the more likely that the system can be validated. Working from a higher level of abstraction will also allow errors in the system to be more easily caught, since developers can better see the "big picture" of the system. In addition to allowing complex systems developers to work at a higher level of abstraction, R2D2C also converts the scenarios into a formal model that can be analyzed for concurrency-related errors and consistency and completeness, as well as domain-specific errors.

Expert Systems

We have been studying the potential use of this approach in the development, maintenance, and verification of expert systems. In particular, we have been giving consideration to using the R2D2C method in verifying the expert system used in the NASA ground control center for the POLAR spacecraft, which performs multi-wavelength imaging of the Earth's aurora. The POLAR ground control expert system has rules written in the production system CLIPS [6] for automated "lights out" (untended) operation of the spacecraft. A suitable translator from CLIPS (rather than natural language) to CSP (or EzyCSP) enables us to use this technology to examine existing expert system rule bases for consistency, etc. What has proven to be of great interest, however, is the ability to generate CLIPS rules from CSP (or EzyCSP), just as we would generate code in Java or C++. POLAR ground control center personnel expect this would be a great benefit because it would give them a means of capturing expert knowledge, from natural language description through to CLIPS rules, while maintaining correctness, which heretofore has not been available.

Robotic Operations

As pointed out earlier, the "code" generated by this approach need not be specifically code in a programming language, and we have been experimenting with generating code to control robots. Perhaps more interesting is the use of this approach to investigate the validity and correctness of procedures for complex robotic assembly or repair tasks in space. We have begun exploratory work in this direction, to provide an additional means to validate procedures from the Hubble Robotic Servicing Mission (HRSM) – for example, the procedures for replacement of cameras on the Hubble Space Telescope (HST).

5. Related Work

Harel [8] [10] has advocated scenario-based programming through UML use cases and play-in scenarios. The present work differs in that it uses scenarios in the form of structured text that is easily understandable by engineers and non-engineers. In addition, the results of converting the structured text to traces and then from traces to a formal model allows us to use a wide range of formal methods tools (e.g., model checkers), which can be used to verify and validate the system.

NASA Ames has been working on the automatic translation of UML use cases to executable code, and report success in using the approach on large applications [26]. Our approach is different, however, in that we are not limited to UML use cases, nor to natural language. R2D2C accommodates any input mechanism whereby requirements can be represented as scenarios, and traces extracted. Our approach works equally well with graphical, mathematical, and textual requirements representations. More importantly, the key to our approach and what makes it invaluable for high-dependability applications is the full formal basis, and complete mathematical tractability from requirements through to code. To our knowledge, no other currently available automated development methodology can make this claim.

6. Conclusions and Future Work

R2D2C is a unique approach to the automatic derivation of ultra-high dependability systems. It is unique in that it supports fully (mathematically) tractable development from

requirements elicitation through to automatic code generation (and back again). While other approaches have supported various subsets of the development lifecycle, there has been heretofore a "jump" in deriving from the requirements the formal model that is a prerequisite for sound automatic code generation. Yet, R2D2C is a simple approach, combining techniques and notations that are well understood, well tried and tested, and trusted. The novelty of the approach, and the part of the approach that achieves continuity in the development process, is the use of a theorem prover to reverse the laws of concurrency, and to achieve levels of inference that would be impossible for a human being to perform on all but trivial systems.

R2D2C (and other approaches that similarly provide mathematical soundness throughout the development lifecycle) will decrease costs and delays for the engineering (and re-engineering) of ultra-high dependability systems through automated development. Such technology will dramatically increase assurance of system success by ensuring that requirements are complete and consistent, implementations are true to the requirements, automatically coded systems are bug-free, and implementation behavior is as expected.

Future work will include improving the quality of the embedding of CSP in ACL2, and optimizing that for efficiency. We plan a plethora of support tools to allow us to easily change the level of abstraction in a formal model, to visualize various system models and changes in those models, and to aid in tracking changes through the development process (or the reverse engineering process). We plan to enhance our existing prototype to support the full version of R2D2C, to make it into a fully functional robust prototype, and to apply it to significant examples.

Acknowledgements

This work was funded in part by the NASA Goddard Space Flight Center Technology Transfer Office. We are grateful to Ted Mecum, Nona Cheeks, Chris Kirkman, Diana Cox, Yvette Conwell-Brown, and Keith Dixon for their support and encouragement. Denis Gracanin (Virginia Tech) and John Erickson (University of Texas at Austin) worked with us on the intermediate method described in Section 3.2 and undertook its implementation as a prototype. The method described in this paper is protected under United States and international patent applications assigned to the United States government.

References

[1] F. L. Bauer. A trend for the next ten years of software engineering. In H. Freeman and P. M. Lewis, editors, *Software Engineering*, pages 1–23. Academic Press, 1980.

[2] J. P. Bowen and M. G. Hinchey. Seven more myths of formal methods. *IEEE Software*, 12(4):34–41, 1995.

[3] M. J. Butler. *csp2B : A Practical Approach To Combining CSP and B*. Declarative Systems and Software Engineering Group, Department of Electronics and Computer Science, University of Southampton, February 1999.

[4] S. A. Curtis, J. Mica, J. Nuth, G. Marr, M. Rilee, and M. Bhat. ANTS (autonomous nano-technology swarm): An artificial intelligence approach to asteroid belt resource exploration. In *Proc. Int'l Astronautical Federation, 51st Congress*, October 2000.

[5] Formal Systems (Europe), Ltd. *Failures-Divergences Refinement: User Manual and Tutorial*, 1999.

[6] J. C. Giarratano. *CLIPS User's Guide*. NASA Johnson Space Center, Houston, Texas, 1992.

[7] J. A. Hall. Seven myths of formal methods. *IEEE Software*, 7(5):11–19, 1990.

[8] D. Harel. From play-in scenarios to code: An achievable dream. *IEEE Computer*, 34(1):53–60, 2001.

[9] D. Harel. Comments made during presentation at "Formal Approaches to Complex Software Systems" panel session. *ISoLA-04 First International Conference on Leveraging Applications of Formal Methods*, Paphos, Cyprus. 31 October 2004.

[10] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, 2003.

[11] M. Hinchey, J. Rash, and C. Rouff. Verification and validation of autonomous systems. In *Proc. SEW-26, 26th Annual NASA/IEEE Software Engineering Workshop*, pages 136–144, Greenbelt, MD, November 2001. NASA Goddard Space Flight Center, IEEE Computer Society Press.

[12] M. G. Hinchey and J. P. Bowen, editors. *Applications of Formal Methods*. Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, and Hemel Hempstead, UK, 1995.

[13] M. G. Hinchey and J. P. Bowen, editors. *Industrial-Strength Formal Methods in Practice*. FACIT Series. Springer-Verlag, London, UK, 1999.

[14] M. G. Hinchey and S. A. Jarvis. *Concurrent Systems: Formal Development in CSP*. International Series in Software Engineering. McGraw-Hill International, London, UK, and New York, NY, 1995.

[15] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978.

[16] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall International, Englewood Cliffs, NJ, and Hemel Hempstead, UK, 1985.

[17] IFAD. The vdm++ toolbox user manual. Technical report, IFAD, 2000.

[18] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-Aided Reasoning: An Approach*. Advances in Formal Methods Series. Kluwer Academic Publishers, Boston, 2000.

[19] K. Lano and H. Haughton. *Specification in B: an Introduction Using the B-Toolkit*. Imperial College Press, London, UK, 1996.

[20] The MathWorks, Inc., Natick, Massachusettes. *Getting Started with MATLAB*, 2000.

[21] F. McLoughlin. ADaGGIO–an automated directed graph diagramming tool. Master's thesis, Department of Computer Science and Information Systems, University of Limerick, Limerick, Ireland, 1992.

[22] D. L. Parnas. Using mathematical models in the inspection of critical software. In *Applications of Formal Methods*, International Series in Computer Science, pages 17–31. Prentice Hall, Englewood Cliffs, NJ, and Hemel Hempstead, UK, 1995.

[23] J. L. Rash, M. G. Hinchey, C. A. Rouff, D. Gracanin, and J. D. Erickson. Experiences with a requirements-based programming approach to the development of a NASA autonomous ground control system. In *EASe, 2nd IEEE Workshop on Engineering of Autonomic Systems, Proc. ECBS 2005, 12th IEEE International Conference on Engineering of Computer-Based Systems*, Greenbelt, MD, 4–7 April 2005.

[24] C. Rouff, J. Rash, and M. Hinchey. Experience using formal methods for specifying a multi-agent system. In *Proc. Sixth IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2000)*, Tokyo, Japan, 2000. IEEE Computer Society Press.

[25] H. J. van Zuylen. *The REDO Compendium: Reverse Engineering for Software Maintenance*. John Wiley and Sons, London, UK, 1993.

[26] J. Whittle, J. Saboo, and R. Kwan. From scenarios to code: An air traffic control case study. In *Proc. ICSE-25, 25th IEEE/ACM International Conference on Software Engineering*, pages 490–495, Portland, Oregon, 2003. IEEE Computer Society Press.