A Reference Model for Requirements and Specifications

Carl A. Gunter
University of Pennsylvania

Elsa L. Gunter
Bell Labs

Michael Jackson
AT&T Laboratories

Pamela Zave *
AT&T Laboratories

March 15, 2000

Keywords: software engineering methodology, refinement, formal methods, requirements, specifications.

Abstract

We define a reference model for applying formal methods to the development of user requirements and their reduction to behavioral specification of a system. The approach is characterized by its focus on the shared phenomena that define the interface between the system and the environment in which it will operate and on how the parts of this interface are controlled. This paper extends our previous work on this model by representing it in higher-order logic and determining some of its key mathematical ramifications. In particular, we introduce a new form of refinement which is pivotal to defining the desired soundness and consistency properties precisely.

1 Introduction

There are a collection of artifacts that commonly arise in programming projects, including the program, and the document that describes the requirements of the software. This requirements document may undergo many revisions as the project proceeds. Requirements often fall into two categories: those intended to be understood by people who commission, pay for, or use the software, and those intended to communicate to programmers what must be coded. These documents are sometimes distinct, receiving distinguishing names like 'customer specification document' versus 'feature specification document'. The distinction also appears in standards; for instance, the software standard of the European Space Agency [10] distinguishes between the 'User Requirements Specification' and the 'Software Requirements Specification', mandating complete documentation of each according to various rules. In other cases this distinction is less emphasized. For instance [9], which discusses software engineering for some of the groups at Microsoft, argues that the difficulty of keeping a technical specification consistent with the program is more trouble than the benefit merits. A wide range of views can be found in the literature and the many organizations that write software.

Is it possible to bring these various artifacts into greater relief and study their properties in a general way, given the wide variations in the use of terms and the many different kinds of software being written? In this paper we attempt to describe a *reference model* for certain key artifacts arising in software projects. The aim is to provide a framework for talking about these artifacts, their attributes and relationships at a general level, but precisely enough that one can rigorously analyze substantive properties. Reference models have a time-honored status in computer science. One very well-known example is the ISO 7-Layer Reference Model, which divides network protocols into seven layers. The model is informal, and does not correspond perfectly to the protocol layers in widespread use, but it still is discussed in virtually every basic textbook on networks, and the model itself is very widely used to describe network architectures. The ISO 7-layer model was successful because it drew on what was already understood about networks and was made general enough to be very flexible. We hope the reference model we describe can provide some of these kinds of benefits to software engineering.

Our model is based on five familiar artifacts classified broadly into those that pertain mostly to the system versus those that pertain mostly to the environment. These artifacts are:

Domain Knowledge provides presumed facts about the environment,

Requirements indicate what the customers need from the system, described in terms of its effect in the environment,

Specifications provide enough information for a programmer to build a system to satisfy the requirements,

Program implements the specification on the programming platform,

Programming Platform provides the basis for programming a system to satisfy the requirements and specifications.[1]

If these are denoted W, R, S, P, and M respectively, then their classification is given by the top Venn diagram in Figure 1. (W and M were chosen for "world" and "machine".) Most of this paper will discuss the special role of the specification, S, which occupies the middle ground between the system and its environment. We hope to carry out a formal analysis, describing the relations that S must satisfy, and

*Email: gunter@cis.upenn.edu, elsa@research.bell-labs.com, jacksonma@acm.org, pamela@research.att.com.

[1] Sometimes the system is construed to include such things as the procedures employed by people who use the software. In this case the people are also programmable and their program is this set of procedures. In this paper we will focus primarily on programming computer platforms.

a comparison of this work to similar attempts to formally analyze this interface.

The paper is divided into six sections. After this introduction we describe the concepts of designation and control (Section 2) in preparation for laying out the key proof obligations of the model (Section 3). These obligations are key to the meanings of the components and are just as important and useful for informal applications of the reference model as they are for formal ones. We then discuss the role of specifications as the bridge between environment and system, again giving the fundamental proof obligations (Section 4). Section 5 discusses related work, including a detailed comparison of our model with the well-known Functional Documentation Model [12, 11] (sometimes called the 'Four Variables Model'). Finally, there is a brief set of conclusions (Section 6).

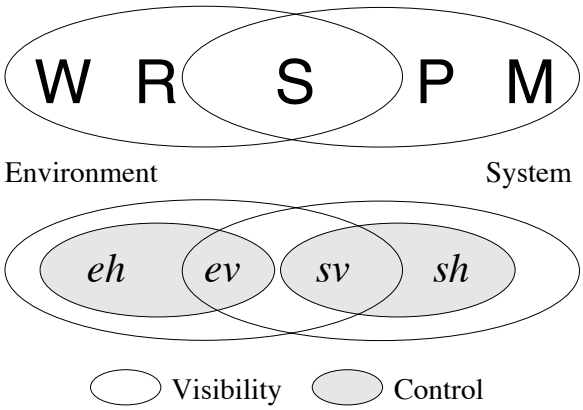

Figure 1: Five Software Artifacts with Visibility and Control for Designated Terms

2 Designations

The WRSPM (Figure 1) artifacts may be viewed primarily as descriptions written in various languages, each based on its own vocabulary of primitive terms. Some of these terms will be shared between one or more of the WRSPM descriptions. To understand the relationships between the WRSPM descriptions, it is essential to understand how the division between environment and system is reflected in the terms used in them. This will determine the key concept of *control*, which will form the basis of a theory of *refinement* described in a later section. This theory of refinement is the basic set of relations between the artifacts.

The distinction between environment and system is a classic engineering issue which is sometimes regarded as a matter of taste and convenience but has a profound effect on the analysis of a problem. The reference model demands a clarification of the primitive terms that are used in the WRSPM artifacts. This clarification is so important that it should be viewed as a sixth artifact in the reference model: the *designated terminology* provides names to describe the application domain (environment), the programming platform with its software (system) and the interface between them.

Designations identify classes of phenomena—typically states, events and individuals—in the system and the environment, and assign formal terms (names) to them. Some of these phenomena belong to the environment and are controlled by it: we will denote this set by e. Others belong to the system and are controlled by it: we will denote this set by s.

At the interface between the environment and the system, some of the e phenomena are *visible* to the system: we will denote this subset of e by e_v; its complement in e are *hidden* from the system, and we will denote this set by e_h. Thus $e = e_h \cup e_v$. The s phenomena are similarly decomposed into s_v and s_h. We assume that e and s are disjoint, an assumption we will analyze later.

Terms denoting phenomena in e_h, e_v and s_v are said to be *visible to the environment*; they are used in W and R. Terms denoting phenomena in s_h, s_v and e_v are said to be *visible to the system*; they are used in P and M. Only e_v and s_v are visible to both the environment and the system; S is restricted to the use of only these terms. The Venn diagram at the bottom of Figure 1 shows the relationships among the four sets of phenomena. A small example will help with understanding some of the ideas in the reference model. We describe a simple version of the *Patient Monitoring System* in our terms. The *requirement* R is a warning system to notify a nurse that the heart-beat of a patient has stopped. To do this, there is a *programming platform* M with a sensor to detect sound on the patient's chest and an actuator which can be *programmed* P to sound a buzzer based on data received from its sensor. There is also some knowledge of the *world* W which says that there is always a nurse close enough to the nurse's station to hear the buzzer, and that if the patient's heart has stopped, then the sound on the patient's chest falls below a threshold for a certain time. The designated terminology falls into four groups, referring to Figure 1.

- e_h: the nurse and the heartbeat of the patient.
- e_v: sounds from the patient's chest.
- s_v: the buzzer at the nurse's station.
- s_h: internal representation of data from the sensor.

The *specification* S, which is expressible in the language common to the environment and system, says that if the sound detected by the sensor falls below the appropriate threshold, then the system should sound the buzzer.

Some notational background is useful for the rest of the paper. The WRSPM reference model is independent of the choice of language for expressing the various artifacts. However, for uniformity, we will assume all of the artifacts are described using formulae of Church's *Higher Order Logic* (HOL). If

$$e_h = \{x_1, \ldots, x_n\},$$

then a formula $\forall e_h.\ \phi$ means the same as $\forall x_1, \ldots, x_n.\ \phi$. We use HOL notational conventions in the paper and hope they are sufficiently obvious that no background is needed beyond what we give here together with some general knowledge of logic. The 'dot' notation requires some care: a dot following a quantification means that the scope of the quantification goes as far to the right as the parentheses allow. For instance $(\exists x.\ A \Rightarrow B) \wedge C$ is the same as $(\exists x.\ (A \Rightarrow B)) \wedge C$.

3 Relationship between Environment and System

The program and the world each have a capacity for carrying out events. The world W restricts the actions that can

be performed by the environment by restricting e or the relationship between e and s_v. The requirements R describe more restrictions saying which of all possible actions are the ones that are desired. The program P, when evaluated on the programming platform M describes the class of possible system events.[2] If the events in this class are all allowed by R, then the program is said to *implement* the requirements. Said in logic, this means:

$$\forall e\ s.\ \mathsf{W} \wedge \mathsf{M} \wedge \mathsf{P} \Rightarrow \mathsf{R} \tag{1}$$

That is, all of the events performed by the environment (e_h, e_v) and all of events (s_v, s_h) performed by the system, taken together, are events allowed by the requirements. We call this property *adequacy*. Adequacy would be trivially satisfied if the assumptions about the environment meant that there is *no* set of events that could satisfy its hypothesis. We therefore need some kind of non-triviality assumption. First of all, we would like *consistency* of the domain knowledge. The desired property is:

$$\exists e\ s.\ \mathsf{W} \tag{2}$$

(Note that this is the same as $\exists\ e_h\ e_v\ sv.\ \mathsf{W}$ since the variables s_h do not appear in W.) Clearly we want consistency of W, P, M together, but there is something more that is needed, a property that says that *any* choice of values for the environment variables visible to the system is consistent with $\mathsf{M} \wedge \mathsf{P}$ if it is consistent with assumptions about the environment.[3] The desired property is called *relative consistency:*

$$\forall e_v.\ (\exists e_h\ s.\ \mathsf{W}) \Rightarrow (\exists e_h\ s.\ \mathsf{W} \wedge \mathsf{M} \wedge \mathsf{P}) \tag{3}$$

Note that the witness to the existential in the conclusion can be the same as the witness in the hypothesis.

Relative consistency merits some appreciation since there are a variety of ways to get the wrong property. It is a significant contribution of the Functional Documentation Model [11], which asserts the relative consistency of the requirements with the domain knowledge. Let $M' = \mathsf{M} \wedge \mathsf{P}$, and consider the property

$$\exists e\ s.\ \mathsf{W} \wedge M' \qquad \textit{(too weak)}$$

This says that there is *some* choice of the environment events that makes the system consistent with the environment. Clearly this is too weak, since the environment may not be so obliging as to use only this consistent set of events. However, this formula clearly should hold, and it does indeed follow immediately from consistency of the domain knowledge (Formula 2) and relative consistency (Formula 3). The formula

$$\forall e\ s.\ \mathsf{W} \Rightarrow M' \qquad \textit{(too strong)}$$

is much too strong since it means that *any* choice of potential system behavior (s) that W accepts must also be accepted by the system to be built (M'). An apparently modest weakening:

$$\forall e\ \exists s.\ \mathsf{W} \Rightarrow M' \qquad \textit{(now too weak)}$$

[2]Of course, programs and programming platforms are not usually presented as formulas. Here one should think of P and M as the formulations of these artifacts in logic.

[3]This assumption is too strong for some cases where the system is expected to prevent the environment from doing some events that are controlled by the environment but visible to the system. The precise reformulation of this criteria for such cases is a topic of ongoing research.

is too weak because, given an environment action, it allows the system to do anything it chooses if there is a corresponding value for the system actions that invalidates the domain knowledge. If we go one step closer to Formula 3 with

$$\forall e.\ (\exists s.\ \mathsf{W}) \Rightarrow (\exists s.\ \mathsf{W} \wedge \mathsf{M} \wedge \mathsf{P}) \qquad \textit{(too strong)}$$

we are once again too strong. The environment actions hidden from the system include reactions to the behavior of the machine. With this formula, the machine is not allowed to restrict any of the possible environmental reactions. In the patient monitoring example it is consistent with the domain knowledge for the patient's heart to stop beating (e_v) without the nurse being warned (e_h); indeed, this is the undesirable possibility that a new program is intended to prevent. But the formula above says that, if this can happen, then the program must allow it!

4 Specifications

Let us suppose now that we wish to decompose the process of implementing a requirement into two parts: a first part in which requirements are developed, and a second part in which the programming is carried out. These tasks may be done by two largely different groups of people, the first perhaps being the users and the second being the programmers. It is often desirable to filter out the knowledge of W and R that truly concerns the people who will work on developing P (for the programming platform M) and deliver this as a *specification* of the software to be built. A kind of transitive property is relied upon to ensure the desired conclusion: if S properly takes W into account in saying what is needed to obtain R, and P is an implementation of S for M, then P implements R as desired. There are several reasons for wanting such a factorization. A common one is the need to divide responsibilities in a contract between the needs of the user and supplier: they build their deal around S, which serves as their basis of communication. But how can we represent this precisely? Is it all right just to say that S and W imply R, while M and P imply S? This is close and provides a good intuition, but the situation is not that simple. Consistency and control must be properly accounted for.

Before we begin to describe proof obligations, we make one stipulation: the specification S must lie in the common vocabulary of the environment and system. That is, the free variables of S must be among those in e_v and s_v, and therefore cannot include any of those in e_h or s_h. Since the specification is to stand proxy for the program with respect to the requirements, it clearly should satisfy the basic properties that the program did. First, we require there should be adequacy with respect to S:

$$\forall e\ s.\ \mathsf{W} \wedge \mathsf{S} \Rightarrow \mathsf{R} \tag{4}$$

Second, we require a strengthened version of relative consistency for S:

$$\forall e_v.\ (\exists e_h\ s.\ \mathsf{W}) \Rightarrow (\exists s.\ \mathsf{S}) \wedge (\forall s.\ \mathsf{S} \Rightarrow \exists e_h.\mathsf{W}) \tag{5}$$

These two formulae, together with 2 are the 'environment-side' proof obligations.

On the other side, the specification is to stand proxy for the requirements (and the domain knowledge) with respect to the program. The 'system-side' proof obligation is a similarly strengthened version of relative consistency for $\mathsf{M} \wedge \mathsf{P}$ with respect to S:

$$\forall e.\ (\exists s.\ \mathsf{S}) \Rightarrow (\exists s.\ \mathsf{M} \wedge \mathsf{P}) \wedge (\forall s.\ (\mathsf{M} \wedge \mathsf{P}) \Rightarrow \mathsf{S}) \tag{6}$$

In summary, if the 'buyer' (of the software) is responsible for the environment-side obligations and the 'seller' is responsible for the system-side obligations, then the buyer must satisfy 2, 4 and 5, and the seller must satisfy 6. The fundamental obligations described by 1, 2 and 3 follow from this. We omit detailed proof of this, but 1 is a consequence of 5 and 6. Formula 3 is a consequence of 4, 5 and 6. Formula 2 was a direct responsibility of the buyer.

We have refered to Formulae 5 and 6 as strengthened versions of relative consistency. However, if they were not strengthened, what we would expect for 5, by comparison with Formula 3, would be

$$\forall e_v.\ (\exists e_h\ s.\ \mathsf{W}) \Rightarrow (\exists e_h\ s.\ \mathsf{W} \wedge \mathsf{S}). \tag{7}$$

This is a direct consequence of Formula 5. The strengthening comes in the added constraint that $\forall s.\ \mathsf{S} \Rightarrow \exists e_h.\ \mathsf{W}$ instead of simply having $\exists e_h\ s.\ \mathsf{W} \wedge \mathsf{S}$. The added constraint means that W must hold everywhere that S holds. The weaker constraint only requires that there is somewhere that they both hold.

It might seem that the weaker Formula 7 should suffice in place of Formula 5. To see why it does not, consider a "good" specification S_1 that satisfies 5 and guarantees R. But suppose that we are also given a "bad" specification S_2 that is everywhere inconsistent with W (see Section 5 for an example). If we let $\mathsf{S} = S_1 \vee S_2$, then S satisfies Formula 4 and the weaker Formula 7, but not Formula 5. An implementor could satisfy his obligation 7 by implementing S_2. Yet an implementation of S_2 would behave unpredictably when deployed. The extra strength of 5 and 6 prevent this.

5 Related Work

The reference model is a more formal and more complete version of some of our earlier work [6, 7, 13]. Let us look in some detail at some of the most well-known formulations of something like the WRSPM artifacts and their relationships.

In the Functional Documentation model [12, 11] there are four distinct collections of variables: m for monitored values, c for system-controlled values, i for values input to the program's registers, and o for values written to the program's output registers. There are also five predicates formally representing the necessary documentation: $\mathsf{NAT}(m,c)$ describing nature without making any assumptions about the system, $\mathsf{REQ}(m,c)$ describing the desired behavior of the system, $\mathsf{IN}(m,i)$ relating the monitored real-world values to their corresponding internal representation, $\mathsf{OUT}(o,c)$ relating the software-generated outputs to external system-controlled values, and $\mathsf{SOF}(i,o)$ relating program inputs to program outputs.

There are three major proof obligations in the Functional Documentation model. The first is called feasibility, and requires that

$$\forall m.\ (\exists c.\ \mathsf{NAT}(m,c)) \Rightarrow (\exists c.\ \mathsf{NAT}(m,c) \wedge \mathsf{REQ}(m,c)).$$

The second states that IN must handle all cases possible under NAT:

$$\forall m.\ (\exists c.\ \mathsf{NAT}(m,c)) \Rightarrow (\exists i.\ \mathsf{IN}(m,i)).$$

The third is called acceptability, and requires that

$$\forall m\ i\ o\ c.\ \mathsf{NAT}(m,c) \wedge \mathsf{IN}(m,i) \wedge \mathsf{SOF}(i,o) \wedge \mathsf{OUT}(o,c) \Rightarrow \mathsf{REQ}(m,c).$$

Although widely accepted, these proof obligations are not sufficient, as can be seen from a small example. Let all the variables be real-valued functions of time, and let the five predicates be defined as:

$$\begin{aligned}
&\mathsf{NAT}: \ (\forall\ t.\ c(t) > 0) \wedge (\forall\ t.\ m(t) < 0)\\
&\mathsf{REQ}: \ \forall\ t.\ c(t+3) = -m(t)\\
&\mathsf{IN}: \ \forall\ t.\ i(t+1) = m(t)\\
&\mathsf{SOF}: \ \forall\ t.\ o(t+1) = i(t)\\
&\mathsf{OUT}: \ \forall\ t.\ c(t+1) = o(t)
\end{aligned}$$

Each predicate is internally consistent. All the predicates besides NAT are readily implementable, since all establish relationships between their inputs at one time and their outputs at a later time. The predicates satisfy all the proof obligations of the Functional Documentation model, yet they are not realizable because

$$\neg(\exists m\ i\ o\ c.\ \mathsf{NAT}(m,c) \wedge \mathsf{IN}(m,i) \wedge \mathsf{SOF}(i,o) \wedge \mathsf{OUT}(o,c)).$$

If the program flipped the sign of its input to get the delayed output, all would be well. Note that the acceptability obligation is satisfiable only because the antecedent of its implication is always false.

Our reference model supplies what is missing in the Functional Documentation model. To show how, we need to match corresponding parts of the two models. The correspondence is not completely determined because it is not clear whether IN and OUT should be considered parts of the system or parts of the environment. In the following we shall consider them to be parts of the system, but our reference model would still have supplied the missing obligation, even if we had interpreted them as being in the environment instead.

In this context, both the phenomena i and o belong to our category s_h. The monitored phenomena m are the same as our e_v phenomena, the controlled phenomena c are the same as our s_v phenomena, and there are no e_h phenomena in the Functional Documentation model.

The predicate NAT corresponds to our domain knowledge W, and the predicate REQ corresponds to our requirements R. NAT and REQ are more restricted than W and R, however, because they can only make assertions about those phenomena of the environment that are shared with the system. In fact, REQ corresponds to the specification S as well as to R. The predicate SOF corresponds to the program P. IN and OUT together correspond to the programming platform M, except that once again they are more restricted, being limited to the special purposes of sensing and actuating.

Since these correspondences make S irrelevant, we shall use our original proof obligations 1-3. Translated into their terms, our Formula 1 is exactly the same as their acceptability. Our Formula 2 translates to $\exists m\ c.\ \mathsf{NAT}(m,c)$, which is not made explicit in the Functional Documentation model because it is assumed to be true by construction. Our Formula 3 translates to

$$\forall m.\ (\exists c.\ \mathsf{NAT}(m,c)) \Rightarrow (\exists i\ o\ c.\ \mathsf{NAT}(m,c) \wedge \mathsf{IN}(m,i) \wedge \mathsf{SOF}(i,o) \wedge \mathsf{OUT}(o,c)),$$

which would have revealed the programming bug in the example above. It also subsumes their second (unnamed) obligation. Translated into our terms, their feasibility obligation is

$$\forall e_v.\ (\exists e_h\ s_v.\ \mathsf{W}) \Rightarrow (\exists e_h\ s_v.\ \mathsf{W} \wedge R),$$

which is implied by our Formulas 1 and 3.

Like our reference model, the Functional Documentation model insists on a rigid division between system and environment control of designated terminology. Some other systems, like Unity [4] and TLA [8, 1], leave to the user such matters as distinguishing environment from system and domain knowledge from requirements, but support shared control of a designated term by system and environment. For example, in the Unity formalism it is possible to express that both E and M control a Boolean variable b, but E can only set it to true while M can only set it to false. Our restricted notion of control, in which each phenomenon must be strictly environment-controlled or strictly system-controlled, is easier to document and think about than shared control. When necessary, shared control can be modeled by using two variables—one controlled by the system, the other by the environment—together with an assertion in W that they must be equal.

The composition and decomposition theorems of TLA [1] are particularly valuable when parts of the formal documentation are not complete and unconditional (as they are required to be in the Functional Documentation model). For example, the TLA composition theorem can be used to prove adequacy of a specification even when all of the domain knowledge, requirements, and specification components are written in an assumption/guarantee style.

As part of a benchmark problem for studying the reference model, we used the model checker Mocha [2, 3] to prove the desired properties of the relationship between specification and program (that is, Formula 5). The Mocha concept of a *reactive model* is extremely similar to our reference model; reactive models provide for 'interface' variables controlled by the environment or system, and system-controlled variables that are hidden from the environment. Formula 5 was partially inspired by its appropriateness to the reactive module assumptions.

6 Conclusions

The reference model described here is meaningful whether or not one is using a formalization like a theorem prover or a model checker. The proof obligations are just as sensible for natural-language documentation as they are for formal specifications. Moreover, there is no absolute requirement that the proof obligations be met in the sense of automated theorem proving. On the contrary, the applications we studied have benefited significantly just from the clarity of knowing what the objective of a component of the model should be, even without formalization, let alone machine-assisted proof. However, our description is precise enough to support quite formal analyses. For examples of such formal analyses, and a more in-depth discussion of some of the logic connections, see [5]. On the whole we think it makes a useful contribution to understanding software artifacts and methodologies at a quite general level without surrendering mathematical precision.

Acknowledgements

We would like to express thanks to Samson Abramsky, Rajeev Alur, Karthik Bhargavan, Trevor Jim, Insup Lee, and Davor Obradovic for their input to this work. This research was partially supported by NSF Contract CCR9505469 and ARO Contract DAAG-98-1-0466.

References

[1] Martin Abadi and Leslie Lamport. Conjoining specifications. ACM Transactions on Programming Languages and Systems, 17(3):507–534, May 1995.

[2] R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 207–218, 1996.

[3] R. Alur, T.A. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. Mocha: Modularity in Model Checking. In *Proceedings of the Tenth International Conference on Computer Aided Verification*, pages 521–525, 1998.

[4] K. Mani Chandy and Jayadev Misra. *Parallel program design: A foundation*. Addison-Wesley, 1988.

[5] Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. Technical report, University of Pennsylvania, January 2000.

[6] Michael Jackson and Pamela Zave. Domain descriptions. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 56–64. IEEE Computer Society Press, 1992.

[7] Michael Jackson and Pamela Zave. Deriving specifications from requirements: An example. In *Proceedings of the Seventeenth International Conference on Software Engineering*, pages 15–24. IEEE Computer Society Press, 1995.

[8] Leslie Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems, 16(3):872–923, May 1994.

[9] Stephen A. Maguire. *Debugging the Development Process: Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams*. Microsoft Press, 1994.

[10] C. Mazza, J. Fairclough, B. Melton, D. de Pablo, A. Scheffer, and R. Stevens. *Software Engineering Standards*. Prentice Hall, 1994.

[11] David Lorge Parnas and Jan Madey. Functional documentation for computer systems. Science of Computer Programming, 25:41–61, October 1995.

[12] A. John van Schouwen, David Lorge Parnas, and Jan Madey. Documentation of requirements for computer systems. In *Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 198–207. IEEE Computer Society Press, 1992.

[13] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. ACM Transactions on Software Engineering and Methodology, 6(1):1–30, January 1997.

Biographies

Carl A. Gunter is an Associate Professor at the University of Pennsylvania. He does research, teaching, and consulting in programming languages, software engineering, networks,

and security. He is the author of a textbook on the semantics of programming languages.

Elsa Gunter received her Ph.D. from the University of Wisconsin–Madison. She was a Post-doctoral research assistant at both Cambridge University and the University of Pennsylvania. She has been a member of technical staff at Bell Laboratories since 1991.

Michael Jackson has been active in software development methodology for almost forty years. Having established his own company in 1970 and run it for many years, he now works as an independent consultant in London, and as a part-time researcher at AT&T Research in Florham Park, NJ.

Pamela Zave received a Ph.D. from the University of Wisconsin–Madison. After serving on the faculty of the University of Maryland, she joined AT&T Laboratories, where she is now a member of the Network Services Research Laboratory.