A Reference Model for Requirements and Specifications

The authors define a reference model for applying formal methods to the development of user requirements and their reduction to a behavioral system specification. The approach focuses on the shared phenomena that define the interface between the system and environment.

Carl A. Gunter, *University of Pennsylvania*
Elsa L. Gunter, *Bell Labs, Lucent Technologies*
Michael Jackson and Pamela Zave, *AT&T Laboratories*

Requirements for software often fall into two categories: for those who commission, pay for, or use it and for those who program it. These documents are sometimes distinct, receiving distinguishing names such as customer-specification document and feature-specification document. The distinction also appears in standards; for example, the European Space Agency software standard[1] distinguishes between the user-requirements specification and the software-requirements specification, mandating complete documentation of each according to various rules. Other cases emphasize this distinction less. For instance, some groups at Microsoft argue that the difficulty of keeping a technical specification consistent with the program is more trouble than the benefit merits.[2] We can find a wide range of views in industry literature and from the many organizations that write software.

Is it possible to clarify these various artifacts and study their properties, given the wide variations in the use of terms and the many different kinds of software being written? Our aim is to provide a framework for talking about key artifacts, their attributes, and relationships at a general level, but precisely enough that we can rigorously analyze substantive properties.

The Reference Model

Reference models have a time-honored status, and one 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 is still discussed in virtually every basic textbook on networks—the model is widely used by network engineers to describe network architectures. The ISO 7-layer model is successful because it draws on what was already understood about networks, and it's general enough to be flexible.

0740-7459/00/$10.00 © 2000 IEEE

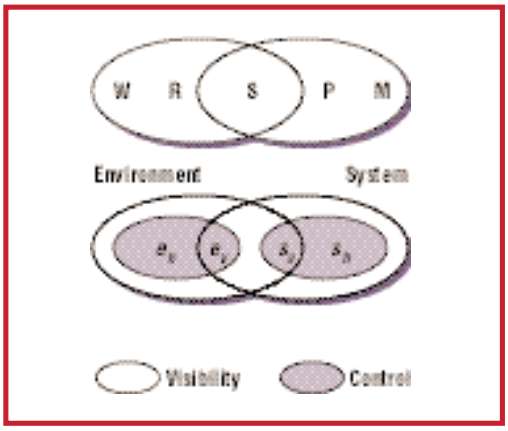

Figure 1. Five software artifacts with visibility and control for designated terms.

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

- domain knowledge that provides presumed environment facts;
- requirements that indicate what the customer needs from the system, described in terms of its effect on the environment;
- specifications that provide enough information for a programmer to build a system that satisfies the requirements;
- a program that implements the specification using the programming platform; and
- a programming platform that provides the basis for programming a system that satisfies the requirements and specifications. (Sometimes the system is construed to include such things as the procedures people who use the software employ. In this case, the people are also programmable, and their program is this set of procedures. Our model primarily focuses on programming computer platforms.)

We denote these artifacts W, R, S, P, and M, respectively, and give their classification in Figure 1's top Venn diagram. (We chose W and M for "world" and "machine.") Most of our discussion focuses on the special role of the specification, S, which occupies the middle ground between the system and its environment. We give a formal analysis, describing the relations that S must satisfy and a comparison of this work to similar attempts to formally analyze this interface.

Designations

We can view the WRSPM artifacts (see Figure 1) primarily as descriptions written in various languages, each based on its own vocabulary of primitive terms. Some of these terms are common to two or more of the WRSPM descriptions. To understand the relationships between the descriptions, we must 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 the *refinement theory*, which is the basic set of relations between the artifacts we describe later.

The distinction between environment and system is a classic engineering issue that is sometimes regarded as a matter of taste and convenience but which has a profound effect on problem analysis. The reference model demands a clarification of the primitive terms that we use in the WRSPM artifacts. This clarification is so important that it is the 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 e. Others belong to the system and are controlled by it; we will denote this set s.

At the interface between the environment and the system, some of the e phenomena are visible to the system: we will denote this e subset e_v. Its complement in e are hidden from the system: 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 disjointed—an assumption we will analyze later.

Terms denoting phenomena in e_h, e_v, and s_v are visible to the environment and used in W and R. Terms denoting phenomena in s_h, s_v, and e_v are visible to the system and used in P and M. Therefore, only e_v and s_v are visible to both the environment and the system. We restrict S to using 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 that notifies a nurse if the patient's heartbeat stops. To do this, there is a programming platform M with a sensor to detect sound in the patient's chest and an actuator that 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, the sound from the patient's chest falls below a threshold for a certain time. The designated terminology falls into four groups (see 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 expressed in the language common to the environment and system, says that if the sound the sensor detects falls below the appropriate threshold, then the system should sound the buzzer.

The WRSPM reference model is independent of the choice of language for expressing the various artifacts. However, for uniformity, we assume all the artifacts are described using formulas of Church's *higher-order logic*. 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 article and hope they are sufficiently obvious that readers don't need background beyond what we give here together with some general knowledge of logic. In the "dot" notation, a dot following a quantification means that the quantification's scope goes as far to the right as the parentheses allow. For instance, $(\exists x.\ A \Rightarrow B) \wedge C$ is the same as $(\exists x.\ (A \Rightarrow B)) \wedge C$.

Relationship between Environment and System

The program and the world each have a capacity for carrying out events. W restricts the actions that the environment can perform by restricting e or the relationship between e and s_v. The requirements, R, describe more restrictions, saying which of all possible actions are desired. P, when evaluated on M, describes the class of possible system events. (Of course, we do not usually present programs and programming platforms as formulas. Here, we should think of P and M as the formulations of these artifacts in logic.) If R allows all the events in this class, then the program is said to implement the requirements. In logic, this means

$$\forall e\ s.\ \mathrm{W} \wedge \mathrm{M} \wedge \mathrm{P} \Rightarrow \mathrm{R}\ . \qquad (1)$$

That is, the requirements allow all the events the environment performs (e_h, e_v) and all the events the system performs (s_v, s_h) that can happen simultaneosly.

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 nontriviality assumption. First, we want *consistency* of the domain knowledge. The desired property is

$$\exists e\ s.\ \mathrm{W}\ . \qquad (2)$$

(This is the same as $\exists\ e_h\ e_v\ s_v.\ \mathrm{W}$, because the variables s_h do not appear in W.) Clearly, we want consistency of W, P, and M together. However, we need something more: a property that says that any choice of values for the environment variables visible to the system is consistent with $\mathrm{M} \wedge \mathrm{P}$ if it is consistent with assumptions about the environment. (This assumption is too strong for some cases where we expect the system 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.) The desired property is called *relative consistency*:

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

The witness to the existential formula in the conclusion can be the same as the witness in the hypothesis.

Relative consistency merits some appreciation, because there are a variety of ways to get the wrong property. It is a significant contribution of the Functional Documentation Model,[3] which asserts the relative consistency of the requirements with the domain knowledge. Let $M' = \mathrm{M} \wedge \mathrm{P}$, and consider the property $\exists e\ s.\ \mathrm{W} \wedge M'$.

This says that there is some choice of the environment events that makes the system consistent with the environment. This is too weak, because the environment might not use only this consistent set of events. However, this formula should hold, and it does immediately follow from the domain-

The WRSPM reference model is independent of the choice of language for expressing the various artifacts.

Because the specification is to stand proxy for the program with respect to the requirements, it should satisfy the same basic properties as the program.

knowledge consistency (Formula 2) and relative consistency (Formula 3). $\forall e\ s.\ W \Rightarrow M'$ is much too strong, because it means that any choice of potential system behavior (s) that W accepts, the system to be built (M') must also accept. An apparently modest weakening, $\forall e\ \exists\ s.\ W \Rightarrow M'$, is too weak, because given an environment action, it lets the system do anything it chooses if there is a corresponding value for the system actions that invalidates the domain knowledge.

One step closer to Formula 3, $\forall e.\ (\exists\ s.\ W) \Rightarrow (\exists\ s.\ W \wedge M \wedge P)$, is again too strong. The environment actions hidden from the system include reactions to the machine's behavior. 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)—this is the undesirable possibility that the program is supposed to prevent. The formula states that, if this can happen, then the program must allow it!

Specifications

Let's suppose now that we wish to decompose the process of implementing a requirement into two parts: first, when requirements are developed, and second, when the programming is carried out. Two different groups of people might do these tasks: users and programmers, respectively. 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 M) and deliver this as a software specification. We rely on a kind of transitive property 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 factorization—for example, 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. We must properly account for consistency and control.

Before we begin to describe proof obligations, we make one stipulation: S must lie in the environment and system's common vocabulary. In other words, 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. Because the specification is to stand proxy for the program with respect to the requirements, it should satisfy the same basic properties as the program. First, we require there to be adequacy with respect to S:

$$\forall e\ s.\ W \wedge S \Rightarrow R\ . \qquad (4)$$

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

$$\forall e_v.\ (\exists e_h\ s.\ W) \Rightarrow (\exists s.\ S) \wedge (\forall s.\ S \Rightarrow \exists e_h.\ W)\ . \qquad (5)$$

These two formulas, together with Formula 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 $M \wedge P$ with respect to S:

$$\forall e.\ (\exists s.\ S) \Rightarrow (\exists s.\ M \wedge P) \wedge (\forall s.\ (M \wedge P) \Rightarrow S)\ . \qquad (6)$$

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

We have referred to Formulas 5 and 6 as strengthened versions of relative consistency. Without the strengthening, what we would expect for Formula 5, by comparison with Formula 3, would be

$$\forall e_v.\ (\exists e_h\ s.\ W) \Rightarrow (\exists e_h\ s.\ W \wedge S) \qquad (7)$$

which is a direct consequence of Formula 5. The strengthening comes in the added con-

straint that $\forall s.\ S \Rightarrow \exists e_h.\ W$, instead of simply having $\exists e_h.\ s.\ W \wedge S$. 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 Formula 5 and guarantees R. But suppose that we are also given a "bad" specification S_2 that is everywhere inconsistent with W. If we let $S = S_1 \vee S_2$, then S satisfies Formula 4 and the weaker Formula 7, but not Formula 5. An implementor could satisfy his obligation Formula 7 by implementing S_2. Yet an implementation of S_2 would behave unpredictably when deployed. The extra strength of Formulas 5 and 6 prevent this.

Comparing Approaches

Our reference model is a more formal and complete version of some of our earlier work.[4–6] Let's look in some detail at some of the most well-known formulations similar to the WRSPM artifacts and their relationships.

In the functional-documentation model,[3,7] 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: NAT(*m*, *c*) describing nature without making any assumptions about the system, REQ(*m*, *c*) describing the desired system behavior, IN(*m*, *i*) relating the monitored real-world values to their corresponding internal representation, OUT(*o*, *c*) relating the software-generated outputs to external system-controlled values, and 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 $\forall m.\ (\exists c.\ \mathrm{NAT}(m, c)) \Rightarrow (\exists c.\ \mathrm{NAT}(m, c) \wedge \mathrm{REQ}(m, c))$. The second states that IN must handle all cases possible under NAT: $\forall m.\ (\exists c.\ \mathrm{NAT}(m, c)) \Rightarrow (\exists i.\ \mathrm{IN}(m, i))$. The third is called acceptability, and requires that $\forall m\ i\ o\ c.\ \mathrm{NAT}(m, c) \wedge \mathrm{IN}(m, i) \wedge \mathrm{SOF}(i, o) \wedge \mathrm{OUT}(o, c) \Rightarrow \mathrm{REQ}(m, c)$.

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

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

Each predicate is internally consistent. All the predicates besides NAT are readily implementable, because 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 $\neg(\exists m\ i\ o\ c.\ \mathrm{NAT}(m, c) \wedge \mathrm{IN}(m, i) \wedge \mathrm{SOF}(i, o) \wedge \mathrm{OUT}(o, c))$. If the program flipped the sign of its input to get the delayed output, all would be well. 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 must 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 the system or environment parts. We consider them system parts, but our reference model still supplies the missing obligation, even if we interpret 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, the controlled phenomena *c* are the same as our s_v, and there are no e_h phenomena in the functional-documentation model.

The predicate NAT corresponds to our W, and the predicate REQ corresponds to our 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 M, except that once again they are more restricted, being limited to the special purposes of sensing and actuating.

Because these correspondences make S irrelevant, we shall use our original Formulas 1 through 3. Translated into their terms, our

Our reference model supplies what is missing in the functional documentation model.

The applications we studied have benefited significantly just from the clarity of knowing what the objective of a model's component should be, even without formalization.

Formula 1 is exactly the same as their acceptability. Our Formula 2 translates to $\exists m\ c.$ 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 $\forall m.\ (\exists c.\ \text{NAT}(m, c)) \Rightarrow (\exists i\ o\ c.\ \text{NAT}(m, c) \wedge \text{IN}(m, i) \wedge \text{SOF}(i, o) \wedge \text{OUT}(o, c))$, which would have revealed the programming bug in the earlier example. It also subsumes their second (unnamed) obligation. Translated into our terms, their feasibility obligation is $\forall e_v.\ (\exists e_h\ s_v.\ \text{W}) \Rightarrow (\exists e_h\ s_v.\ \text{W} \wedge R)$, 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, such as Unity[8] and TLA,[9,10] 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, we can 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 or system controlled, is easier to document and think about than shared control. When necessary, we can model shared control by using two variables—one controlled by the system and the other by the environment—together with an assertion in W that they must be equal.

The composition and decomposition theorems of TLA are particularly valuable when parts of the formal documentation are not complete and unconditional (as the functional-documentation model requires them to be). For example, we can use the TLA composition theorem to prove adequacy of a specification even when all of the domain knowledge, requirements, and specification components are written in an assumption and guarantee style.

As part of a benchmark problem for studying the reference model, we used the model checker Mocha[11,12] to prove the desired properties of the relationship between specification and program (see Formula 5). The Mocha concept of a *reactive model* is extremely similar to our reference model, because it allows for interface variables controlled by the environment or system and system-controlled variables that are hidden from the environment. The model's appropriateness to the reactive module assumptions partially inspired Formula 5.

Our reference model is meaningful whether or not you are using a formalization such as 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 model's component should be, even without formalization, let alone machine-assisted proof. However, our description is precise enough to support formal analyses. For examples of such formal analyses and a more in-depth discussion of some of the logic connections, see our technical report from the University of Pennsylvania.[13]

Our current and future research involves extending and applying the reference model. We are interested in extensions that unify it with game-theoretic modeling of our system and environment. Such extensions might make the model more complex to describe but might offer better guidance when using it for applications. We are applying the model to problems networking and telephony. In networking, we are using it to describe attributes of routing protocols; for telephony, we are using it to model distributed feature composition.

Acknowledgments

We thank Samson Abramsky, Rajeev Alur, Karthik Bhargavan, Trevor Jim, Insup Lee, and Davor Obradovic for their input to this work. NSF Contract CCR9505469 and ARO Contract DAAG-98-1-0466 partially support this research.

References

1. C. Mazza et al., *Software Engineering Standards*, Prentice Hall, Upper Saddle River, N.J., 1994.
2. S.A. Maguire, *Debugging the Development Process: Practical Strategies for Staying Focused, Hitting Ship Dates, and Building Solid Teams*, Microsoft Press, Redmond, Wash., 1994.
3. D.L. Parnas and J. Madey, "Functional Documentation for Computer Systems," *Science of Computer Programming*, Vol. 25, No. 1, Oct. 1995, pp. 41–61.
4. M. Jackson and P. Zave, "Domain Descriptions," *Proc. IEEE Int'l Symp. Requirements Eng.*, IEEE Computer

Soc. Press, Los Alamitos, Calif., 1992, pp. 56–64.
5. M. Jackson and P. Zave, "Deriving Specifications from Requirements: An Example," *Proc. 17th Int'l Conf. Software Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1995, pp. 15–24.
6. P. Zave and M. Jackson, "Four Dark Corners of Requirements Engineering," *ACM Trans. Software Eng. and Methodology*, Vol. 6, No. 1, Jan. 1997, pp. 1–30.
7. A.J. van Schouwen, D.L. Parnas, and J. Madey, "Documentation of Requirements for Computer Systems," *Proc IEEE Int'l Symp. Requirements Eng.*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1992, pp. 198–207.
8. K. Mani Chandy and J. Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, Reading, Mass., 1988.
9. M. Abadi and L. Lamport, "Conjoining Specifications," *ACM Trans. Programming Languages and Systems*, Vol. 17, No. 3, May 1995, pp. 507–534.
10. L. Lamport, "The Temporal Logic of Actions," *ACM Trans. Programming Languages and Systems*, Vol. 16, No. 3, May 1994, pp. 872–923.
11. R. Alur and T.A. Henzinger, "Reactive Modules," *Proc. 11th IEEE Symp. Logic in Computer Science*, IEEE Computer Soc. Press, Los Alamitos, Calif., 1996, pp. 207–218.
12. R. Alur et al., "Mocha: Modularity in Model Checking," *Proc. 10th Int'l Conf. Computer Aided Verification*, A.J. Hu and M.Y. Vardi, eds., *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1998, pp. 521–525.
13. C.A. Gunter et al., *A Reference Model for Requirements and Specifications*, tech. report, Univ. of Pennsylvania, Dept. of Computer and Information Science, Jan. 2000.

About the Authors

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. He received his BA from the University of Chicago and his PhD from the University of Wisconsin, Madison. Contact him at the Dept. of Computer and Information Science, 200 S. 33rd St., Univ. of Pennsylvania, Philadelphia, PA 19104; gunter@cis.upenn.edu.

Elsa L. Gunter is a member of the technical staff at Bell Laboratories. Her research interests include formal methods, design and use of automated and interactive theorem provers, and mathematical semantics of programming languages. She received her BA from the University of Chicago and her PhD from the University of Wisconsin, Madison. She was a postdoctoral research assistant at both Cambridge University and the University of Pennsylvania. Contact her at Bell Labs, Room 2C-481, 600 Mountain Ave, P.O. Box 636, Murray Hill, NJ 07974-0636; elsa@research.bell-labs.com.

Michael Jackson is as an independent consultant in London and a part-time researcher at AT&T Research in Florham Park, N.J. His research interests include requirements specifications, problem analysis, problem frames, software design, and feature interaction. Contact him at 101 Hamilton Terrace, London NW8 9QY, UK; jacksonma@acm.org.

Pamela Zave is a member of the Network Services Research Laboratory. Her research interests include formal methods for telecommunication systems and requirements engineering. She received her AB from Cornell University and her PhD from the University of Wisconsin, Madison. Contact her at AT&T Laboratories–Research, 180 Park Ave., Room D205, Florham Park, NJ 07932; pamela@research.att.com.