# 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

February 1, 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. Among these are the program itself, of course, and also 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 [11] distinguishes between the 'User Requirements Specification' and the 'Software Requirements Specification', mandating complete documentation of each according to various rules. In other cases, by contrast, this distinction is less emphasized. For instance [10], 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 what one might call 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 one will still find it 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

*Email: gunter@cis.upenn.edu, elsa@research.bell-labs.com, jacksonma@acm.org, pamela@research.att.com. This work was funded in part by NFS grant #CCR9505469.

[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.

system and its environment. We hope to carry out a formal analysis, describing the relations that S must satisfy, and 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 [13, 12] (sometimes called the 'four variables' model). Finally, there is a brief set of conclusions (Section 6).

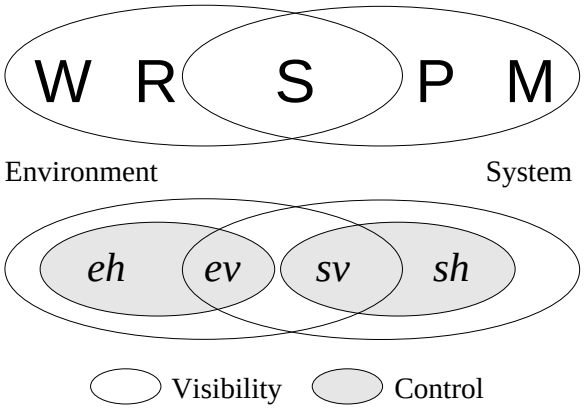


Figure 1: Five Software Artifacts with Visibility and Control for Designated Terms

## 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 $ev$; its complement in $e$ are *hidden* from the system, and we will denote this set by $eh$. The $s$ phenomena are similarly decomposed into $sv$ and $sh$.

Terms denoting phenomena in $eh$, $ev$ and $sv$ are said to be *visible to the environment*; they are used in W, R and S. Terms denoting phenomena in $sh$, $sv$ and $ev$ are said to be *visible to the system*; they are used in S, P and M. 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 the 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 a buzzer sounded there, 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.

- System hidden (that is, hidden from the system) and environment controlled ($eh$): the nurse and the heartbeat of the patient.
- System visible and environment controlled ($ev$): sounds from the patient's chest.
- Environment visible and system controlled ($sv$): the buzzer at the nurse's station.
- Environment hidden and system controlled ($sh$): 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 from the sensor falls below the appropriate threshold, then the system should sound the buzzer.

Some notational background is essential for the rest of the paper. While this reference model is independent of the choice of language for expressing the various artifacts, for uniformity here, and because it is expressive enough, we wiil assume all formulae are formulae of Church's *Higher Order Logic* (HOL) [5]. If

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

then a formula $\forall ev.\ \phi$ means the same as $\forall x_1, \ldots, x_n.\ \phi$ for some ordering of the variables in $ev$. It is usually not necessary to distinguish between $eh$ and $ev$ directly in our formulas so we will use $e = eh \cup ev$. Similarly we take $s = sh \cup sv$. We assume that $e$ and $s$ are disjoint, an assumption we will analyze later. 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 basic intuition behind our treatment is that the program and the world each have a capacity for carrying out events, or perhaps remaining inert. The world W provides restrictions on the actions that can be performed by the environment. These can be understood as restrictions on $e$ or on the relationship between $e$ and $sv$. The requirements R describe an additional set of restrictions saying which of all possible actions are the ones that are desired. The program P, when evaluated on the programming platform M restricts the class of possible events.[2] If this restriction is to a collection of events 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 ($eh$, $ev$) and all of events ($sv$, $sh$) 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 the domain assumptions to be consistent. The desired property is:

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

and we call this *consistency* (of domain knowledge). (Note that this is the same as $\exists\ eh\ ev\ sv.\ \mathsf{W}$ since the variables $sh$ 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 is consistent with $\mathsf{M} \wedge \mathsf{P}$ if it is consistent with assumptions about the environment. The desired property is called *relative consistency:*

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

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 [12], to which we will compare our work later. 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 following:

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

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:

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

is too weak because, given an environment action, it allows the system to do anything it chooses to if there is a corresponding value for the system actions that invalidate the domain knowledge.

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

## 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 alright 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.

### 4.1 Proof Obligations

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 $ev$ and $sv$, and therefore cannot include any of those in $eh$ or $sh$. 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. That is, there should be adequacy with respect to S:

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

and there should be relative consistency for S:

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

It should be noted that the relative consistency of R with respect to W follows from (4) and (5):

$$\forall e.\ (\exists s.\ \mathsf{W}) \Rightarrow (\exists s.\ \mathsf{W} \wedge \mathsf{R}) \tag{6}$$

The aim now would be to create a 'developer-side' set of criteria which, when taken together with 'user-side' criteria like Formulas 1 and 3, would imply the desired relationship between the requirements and the implementation. It is tempting to derive these conditions by attempting an analogy using Formulas 1 and 3 as a template but shifting one's perspective to view M as analogous to domain knowledge, S as analogous to requirements, and P as analogous to

the specification. The scope of this paper does not permit a detailed treatment of this approach, but suffice it to say that the result is both too strong and too weak. It is too strong because it would demand that the program needs to satisfy various properties even for cases the environmental assumptions view as impossible. It is too weak because it does not ensure the consistency of W and M. A new condition is needed; one that implies Formulas 1 and 3, but makes reasonable assumptions about what must be true of S.

Our investigations have led us to a key set of proof obligations that works well with the case studies we have carried out and also provides a clean logical treatment. Our conditions are essentially a strengthening of relative consistency to insist that they include enough information about the domain knowledge to enable a developer to write an acceptable program. At first blush, one could simply ask that all of W be included in S. Indeed this would work, and the desired transitivity would follow, *provided* the designations used in W are all visible to the system. In essence, the specification S must find a way to use system-visible designations to provide the developer with all of the information that is needed about the assumptions in W. If this cannot be done, it probably means that the programming platform lacks the kinds of inputs (sensors) and outputs (actuators) needed to satisfy the requirements. Our condition replaces relative consistency (Formula 3) with two conditions. The first of these is *environment-side refinement:*

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

which is the proof obligation of those who reduce the requirements and domain knowledge to a specification. The second is *system-side refinement:*

$$\forall e.\ (\exists s.\ \mathsf{S}) \Rightarrow (\exists s.\ \mathsf{M} \wedge \mathsf{P}) \wedge (\forall s.\ (\mathsf{M} \wedge \mathsf{P}) \Rightarrow \mathsf{S}) \tag{8}$$

which is the proof obligation of those who implement the specification.

Formulas 7 and 8 are almost the same as relative consistency except for the added constraints that $\mathsf{S} \Rightarrow \mathsf{W}$ and $\mathsf{P} \Rightarrow \mathsf{S}$. These extra obligations are not just a technical convenience; they are essential for the practical application of the reference model. To see why they are necessary in a concrete way, consider the case where we have a 'good' specification $S_1$ that is relatively consistent with respect to the domain knowledge W, and is adequate to guarantee the requirements R. Now also consider a 'bad' specification $S_2$ that is everywhere inconsistent with the domain knowledge (we will provide an example of this kind in Section 5). If we let $\mathsf{S} = S_1 \vee S_2$, then S is also relatively consistent (Formula 5) and adequate (Formula 4). However, if we turned it over to a programmer who built a system satisfying $S_2$, the system would also satisfy S, but it would break as soon as it was deployed in the intended environment. The extra strength of Formulas 7 and 8 prevent this problem from arising.

### 4.2 Summary

Formulas 1, 2, and 3 are our principle proof obligations. They can be proved by defining a specification S and showing 2, 4, 7 (on the environment side), and 8 (on the system side).

## 5 Related Work

The reference model is a more formal and more complete version of some of our earlier work [7, 8, 14]. Let us look in some detail at some of the most well-known formulations of something like the WRSPM artifacts and their relationships.

There are many similarities between our reference model and the Functional Documentation model of Madey, Parnas, and van Schouwen [13, 12]. Finding a precise comparison between the two is a little tricky because our reference model, for clarity and broadest applicability, demands that there be a sharp dividing line between what exists and what is to be built (from the perspective of the particular project at hand). In the Functional Documentation model, on the other hand, there is a third category representing an intermediate phase of engineering. This third category is occupied by the predicate $\mathsf{IN}(m, i)$, describing the behavior of input or sensor devices in translating monitored quantities $m$ to input values $i$, and by the predicate $\mathsf{OUT}(o, c)$, describing the behavior of output or actuator devices in translating output values $o$ to controlled quantities $c$.

Thus we shall have to make two comparisons, one in which the devices are regarded as part of the system in our reference model, and one in which they are regarded as part of the environment. In both comparisons the Functional Documentation model is more or less a special case of the reference model.

In the Functional Documentation model, 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. If the I/O devices are regarded as part of the system, then both the phenomena $i$ and the phenomena $o$ belong to our category *sh*. The monitored phenomena $m$ are the same as our *ev* phenomena, the controlled phenomena $c$ are the same as our *sv* phenomena, and there are no *eh* phenomena in the Functional Documentation model.

Also in the Functional Documentation model, there are five predicates formally representing the necessary documentation: $\mathsf{NAT}(m, c)$ describing nature without any assumptions about the system, $\mathsf{REQ}(m, c)$ describing the desired behavior of the system (including sensors and displays), $\mathsf{IN}(m, i)$ relating the real-world values monitored to their corresponding internal representation, $\mathsf{OUT}(o, c)$ relating the internal values to be output to the actual values displayed, and $\mathsf{SOF}(i, o)$ representing the program computing outputs from inputs. Still viewing the I/O devices as part of the system, the predicate $\mathsf{NAT}(m, c)$ corresponds to our domain knowledge W, and the predicate $\mathsf{REQ}(m, c)$ 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 $\mathsf{SOF}(i, o)$ corresponds to the program P. $\mathsf{IN}(m, i)$ and $\mathsf{OUT}(o, c)$ together correspond to the programming platform M, except once again they are more restricted, for they are only allowed to indicate the relationship between sensors and internal registers of the program.

Now we consider the proof obligations of the Functional Documentation model. One group says that REQ and IN must cover all situations that are physically possible:

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

These are consequences of relative consistency for S (Formula 5) and relative consistency for $\mathsf{M} \wedge \mathsf{P}$ (Formula 3). There are additional proof obligations of showing that SOF

and OUT must cover all possible situations:

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

These proof obligations are not a part of our reference model, but they are approximated by our proof obligation that the combination of the program and the programming platform be relatively consistent with the domain knowledge (Formula 3). The Functional Documentation model does not actually require this relative consistency. We shall say more on this later.

There are two more proof obligations of the Functional Documentation model: their acceptability obligation

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

is exactly the adequacy of $\mathsf{M} \wedge \mathsf{P}$ (Formula 1), and their feasibility obligation

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

is both the relative consistency of S and R (Formulas 5 and 6), since REQ is both R and S. The consistency of W or NAT (Formula 2) is not mentioned in the Functional Documentation model, presumably because NAT is expected to be expressed in a form that makes its consistency obvious. Our proof obligation of adequacy of S (Formula 4) is vacuous because S is the same as R.

Shifting to the second comparison, in which the devices are regarded as part of the environment, $i$ is exactly the same as our phenomena *ev* and $o$ is exactly the same as our phenomena *sv*. Our phenomena *eh* corresponds to the union of their phenomena $m$ and $c$, but $m$ and $c$ are more restricted because they must be distinct and also 'close' to the system in the sense of having a direct relationship through IN and OUT with the shared phenomena $i$ and $o$.

With this correspondence between the two models, their domain knowledge W must be decomposed into three parts $\mathsf{NAT}(m,c)$, $\mathsf{IN}(m,i)$, and $\mathsf{OUT}(o,c)$. REQ corresponds to R but is more restricted because it can constrain $m$ and $c$ only. $\mathsf{SOF}(i,o)$ corresponds exactly to S.

Concerning the proof obligations of the reference model, even assuming that each part of the Functional Documentation model is consistent by construction, and that all the proof obligations of the Functional Documentation model are satisfied, that is still not enough to guarantee our relative consistency property. In this context the relative consistency of S with respect to W takes the form

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

To see why it is not guaranteed, let all the variables be real-valued functions of time, and let the various predicates be defined as:

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

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

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

(The program needs to flip the sign of the input to get the delayed output, but doesn't.) Note that the acceptability obligation is satisfiable only because the antecedent of its implication is always false.

This example is equally problematic if we view the I/O devices as part of the system, instead of as part of the environment. In that case we do have the relative consistency of the specification. What is violated is the relative consistency of the program and programming platform with the domain knowledge:

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

Once again the example fails to satisfy this.

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 [9, 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 7). 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 7 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 HOL 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 [6]. 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 Rajeev Alur, Karthik Bhargavan, Trevor Jim, Insup Lee, and Davor Obradovic for their input to this work.

## 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] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[6] 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.

[7] 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.

[8] 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.

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

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

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

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

[13] 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.

[14] 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 in the areas of programming languages and software engineering with contributions to the semantics of programming languages, type systems, and the design 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.