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

April 16, 1998

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. We illustrate the consequences of these advances for two benchmark problems and for applications of the model in projects at AT&T and Lucent.

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 have been written before the software was built, revised as coding was underway, and revised again after the software was tested. 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 [14] 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 [13], 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 Venn diagram in Figure 1. 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.

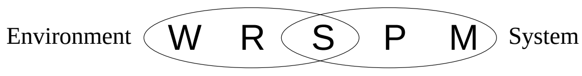

Figure 1: Five Software Artifacts

grounding this analysis in a series of examples, and a comparison of this work to similar attempts to formally analyze this interface.

The paper is divided into nine 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 [17, 15] (sometimes called the 'four variables' model). The next two sections provide case studies, first based on benchmark illustrations (Section 6), and then based on significant projects where we are using the reference model (Section 7). After this there is some analysis (Section 8) and a brief set of conclusions (Section 9).

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 and events and individuals—in the system and the environment, and assign formal terms (names) to them. Some of these phenomena are phenomena belonging to the environment and controlled by it: we will denote this set by e. Some are phenomena belonging to the system and 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. Similarly, at the interface between the environment and the system, some of the s phenomena are *visible* to the environment: we will denote this subset of s by sv; its complement in s are *hidden* from the environment, and we will denote this set by 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. Figure 2 shows the relationships among the four sets of phenomena.

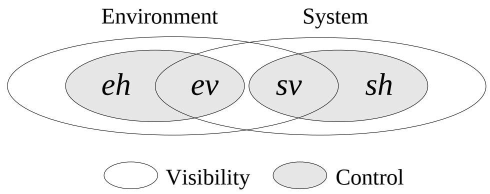

Figure 2: Visibility and Control for the Designated Terms

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 that is able to detect sound on the patient's chest and an actuator capable of sounding a buzzer. This 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 2.

- System hidden (that is, hidden from the system) and environment controlled (eh): the nurse and the heartbeat of the patient.
- System visible (that is, visible to the system) 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.

The treatment so far of the reference model does not mention any particular language in which the artifacts are described. Generally they will be described by different languages. To characterize the relations precisely, however, it will be convenient to use a single language in which others can be embedded. For this we choose a version of Church's *Higher Order Logic*. There are a variety of reasons for the choice, but the most important of these are the fact that it is an expressive language for which there is a great deal of experience with modeling computer systems and embedding languages. There is also automated support in the form of the HOL interactive theorem-proving system [8, 7], which we have used extensively in our experiments. A final reason is that we found the higher-order aspect of the logic convenient for scalability in our applications and helpful in writing down the proof obligations of the reference model.

Some notational background is essential for the rest of the paper. 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 mean 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. Consistency is asserted by existential quantification over the free variables of the formula. 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 [15], 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.

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 being the users (vendees of the software for example) and the second being the programmers (vendors of the software perhaps). 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:

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

[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.

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. This set of conditions is derived from a key relation which we introduce below for those who are interested in the logical essence of the contribution. 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 $\mathsf{P} \Rightarrow S$. 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 A Logical Account

Some basic properties of the proof obligations are essential. We give a brief account, but the reference model can be used both informally and formally without knowing these technical details.

A classic theorem of mathematical logic is the *Craig Interpolation Theorem* (see Theorem 2.2.20 of [6]). It says that if ϕ and ψ are closed first order formulas such that ψ implies ϕ, then there is a closed first order formula θ, *in the common language* of ϕ, ψ such that ψ implies θ and θ implies ψ. The formula θ is called a *Craig Interpolant* for ϕ and ψ. In our context, the specification can be viewed as a common-language interpolant between the environment and system but based on a different concept of interpolation. To define the relation we do use, partition the variables into two disjoint sets e and s. We induce a relation $\geq$ on formulas ϕ, ψ whose free variables lie among $e \cup s$ as follows:

$$\phi \geq \psi \text{ iff } \forall e.\ (\exists s.\ \phi) \Rightarrow (\exists s.\ \psi) \wedge (\forall s.\ \psi \Rightarrow \phi). \tag{9}$$

In this case we say that ψ is a *predicated consistent refinement* of ϕ. It is obvious that this relation is stronger than relative consistency. It is somewhat less obvious that it defines a poset ordering on logical equivalence classes of formulas. First of all, it is clear that $\phi \geq \phi$. Moreover, if $\phi \geq \psi$ and $\psi \geq \phi$, then $\vdash \phi \Leftrightarrow \psi$. But, in particular, the relation is transitive: if $\phi \geq \psi \geq \theta$ then $\phi \geq \theta$. This is the key fact needed to carry out a sequence of refinements without the need for 'global' knowledge of the components. To see this, note that Formula 7 says S is a predicated consistent refinement of W, so Formula 5 follows from this. Formula 8 says that the implementation $\mathsf{M} \wedge \mathsf{P}$ is a predicated consistent refinement of S. By transitivity we therefore learn that the implementation is a predicated consistent refinement of the domain assumptions so we obtain relative consistency of the program with respect to domain knowledge (Formula 3).

4.3 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 [9, 10, 19]. Before looking at applications using the approach described here, we pause to 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 [17, 15]. 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 some more on this later.

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

$$\begin{array}{l} \forall m\ i\ o\ c. \\ \quad \mathsf{NAT}(m, c) \wedge \mathsf{IN}(m, i) \wedge \mathsf{SOF}(i, o) \wedge \mathsf{OUT}(o, c) \\ \qquad \Rightarrow \mathsf{REQ}(m, c) \end{array}$$

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

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

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:

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

Each predicate is consistent, and those that need to be implemented are 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

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

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 [5] and TLA [12, 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:

- b is stable in E
- $\neg b$ is stable in M

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. This advantage is perhaps unimportant for small projects, but it becomes very important in large ones. Also, the restricted notion of control seems sufficient for requirements and specifications. Shared control is most useful for coordinating the actions of closely coupled components in a concurrent algorithm. 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 Illustrations of the Model

This section provides two 'benchmark' applications intended to be fairly simple but illustrate the concepts underlying the reference model. The first of them is a data processing problem and the second is a reactive system.

6.1 The wrap program

Users of a Unix-like operating system, in which a file is a sequence of characters, often print files received as email. If such a file contains long lines, the printing utility will 'print' them off the rightmost edge of the paper, where they are lost to the reader. To prevent this annoyance, the users want a utility program *wrap*. The primary requirement on *wrap* is that, if a file is piped through it before printing, no print line resulting from the file will be longer than 80 characters. Otherwise *wrap* should change its operand as little as possible. This is the user requirement R.

In this example the system ($\mathsf{P} \wedge \mathsf{M}$) is the program *wrap* and its environment is the operating system, particularly the file format and print utility. More specifically, P might be a script, M the semantics of the language in which it was written, and W the assumptions we need to make about printers. To specify *wrap* successfully we need the following knowledge about this printing environment:

- In addition to printing characters, files can contain the non-printing characters newline and tab.
- The print utility recognizes newline characters as commands to start a new print line.
- The print utility replaces the tab character by 8 space characters in the print line.

Armed with this knowledge, we can specify *wrap* as a file transducer S that counts the number of characters in each subsequence delimited by newlines. In this special count, a tab counts as 8 characters, while all other characters count as 1. Wherever this count exceeds 80, *wrap* inserts a newline character.

The primary point of this example is the difference between the requirement and specification for *wrap*. The requirement expresses a constraint on the effect of using *wrap* and *print* together in a certain way, which is what the user cares about. Consequently, the specification of *wrap*—which does not mention *print*—must take into account some knowledge of what *print* does.

A secondary point of this example is that the reference model makes perfect sense even when the 'environment' and 'system' with respect to a particular software-development problem are both embedded in a larger computer system.

6.2 VTS: The Village Telephone System

The Village Telephone System (VTS) is a non-trivial benchmark system that we created for the purpose of experimenting with a fairly complete use of the reference model and our methodology for representing it in HOL. We summarize it here for illustration and refer the reader to [4] for details.

VTS is a service akin to a chat line or an anycast network service. The villagers request a communication system that will allow them to find conversation partners. Rather than Plain Old Telephone Service (POTS), in which one dials a specific number, they would like the system to attempt to find them an arbitrary conversation partner. The villagers do not care who the partner is so long as an attempt is made to find someone. The requirements insist on this 'best effort' of the system, but there can be no guarantee that anyone will answer since all other telephones may be engaged or no one may be home. VTS has a variety of requirements and assumptions about people, sounds, telephones, and communication. These are classified into two modalities: indicative and optative. Indicative assertions are assumptions about the environment independent of the programming of the telephone switching system; these are grouped into W and include assumptions which will be viewed as axioms relative to the proof obligations of the reference model. Optative assertions are the wish list of the villagers concerning their village after the introduction of its communication system. These wishes presumably include objectives that must be satisfied by the selection of a suitable programming platform M and program P. This wish list is the set of requirements R; it is presumably consistent with the indicative assumptions in W. The division between R and W is crucial to our treatment of VTS.

VTS is modeled as a sequence of rounds in which an environmental event occurs and the system responds instantly, possibly by changing its state. Villagers may take phones off-hook or put them on-hook at any time, modulo the assumption (in W) that phones are a toggle. Thus the events of going on or off hook are controlled by the environment. By contrast, the telephone switching system is able to cause phones to enter or end an alerting state (that is, start or stop 'ringing') and create or terminate connections between pairs of telephones. Figure 3 illustrates some of the possibilities.

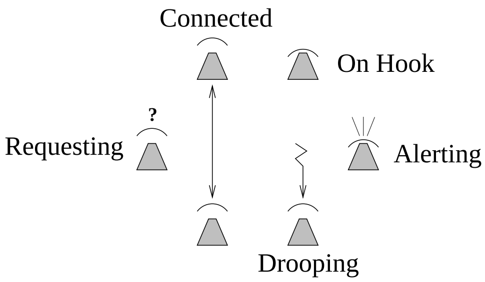

Figure 3: Telephone Events and State

Examples of properties required by the villagers (R) include: off-hook telephone are never alerting, if a villager answers an alerting phone, he will get a connection, villagers can communicate over connected off-hook telephones, connections will only be broken by one of the parties going on hook, and on-hook telephones do not transmit sounds. Examples of properties that are assumed (W) of the telephone system include: connections are in pairs (there are no conference bridges), off-hook connected telephones are capable of transmitting sound, and discconnected telephone are not capable off transmitting sound. Some of the requirements in R follow immediately from assumptions about the domain, which essentially include assumptions about how the programming platform will behave as observed by the environment. Other properties must be satisfied by the programming.

Although it is possible to omit a specification of the VTS and proceed directly to a program that implements the requirements, there is a significant gap between the requirements and the programming platform. For instance the switching system has no direct way of dealing with people or sounds: these are connected to the system through shared phenomena such as on and off-hook events, and the domain knowledge. Moreover, a number of different implementations are possible based on different views of how hard the system should look for another partner. For instance, each telephone could be implemented as a hotline to another telephone, always alerting its partner when taken off-hook in a non-alerting state. At an opposite extreme, an off-hook event for a non-alerting phone could be implemented as a broadcast which causes all on-hook telephones in the village to alert. An intermediate approach is to alert a single on-hook telephone, chosen at random. A variation on this theme is to rotate this choice when no answer is obtained. Another question is the treatment of telephones that are off-hook but holding one end of a broken connection (such telephones are said to be 'drooping', see Figure 3): are such phones now candidates for connection? All of these variations represent possible specifications S, which can then be refined into a specific programming approach, generally laden with more details (for instance, how the selection and rotation is carried out based on primitives of M that are not visible to the environment). Most of these alternatives are consistent with the villagers' requirements, but care must be taken (for instance hotlines is problematic if there are an odd number of telephones), so the specification can serve the role of relieving the writer of P from a need to know about W and R, while not over-binding the form of P.

The specification can also assist the programmer in the detection of dead code and other simplifications and efficiencies. In the VTS example, the most straightforward way to write the specification is to provide an exhaustive case analysis. However, various system invariants mean that many of these cases can never arise. For instance, in one of our specifications there is no more than one alterting telephone at any time, so the case of multiple alerting telephones never arises. Code attempting to treat this case can be viewed as 'dead' since it will never be entered. It is possible to write a logically equivalent specification that does not break out the cases that are impossible, but treats them uniformally with other similar cases that cannot occur. This second specification is best shown to satisify the requirements by proving it is equivalent to the more exhaustive version. The obvious implementations suggested by each of the two specifications are rather different: the most straightforward implementation of the second specification would be more compact because it eliminates some tests and dead code.

7 Applications of the Model

This section described a pair of large scale projects in which we are using the reference model. In both cases we have been able to make effective use of formal support tools for formal descriptions and correctness proofs.

7.1 DSP: Translating DSP Instructions

When the manufacturer of a line of processors makes a change in its instruction set that causes code written for its old chips not to run on its new chip, then a major problem of legacy code arises. If the legacy code was written in a high-level language, then a compiler for the new chip can be written and then the legacy code can be recompiled. Often there are hitches in this approach, but they can usually be patched with less effort than it would take to rewrite the legacy programs for the new chip. This strategy depends on the availability of high-level languages and adequate compilers for the chips in question. Unfortunately there are important classes of processors for which compiler technology is inadequate; in particular, Digital Signal Processors (DSP's) are such an example. DSP's are processors whose instruction sets are highly tuned to signal processing, and are used in devices like mobile telephones. Such applications often have timing constraints that are too strict to make it feasible to use compiled code; the programmer must hand-code many parts of the code, carefully counting the time each instruction will take. This makes DSP code particularly difficult to write, and simultaneously makes it all the more difficult and more important not to lose legacy code because of a shift to a new architecture.

One approach to maintaining legacy code for a class of DSPs is to provide a translator from assembly language programs for the old processor to assembly language programs for the new one. A minimal requirement for such a translator is that the translation of a 'well-constructed' program has the same observable behavior on memory when run on the new chip as the original program has when run on the old chip provided they are run in comparable well-formed initial states. Then, as a separate step, one can use various techniques to determine whether the translation satisfies the necessary timing constraints. (This might be obtained 'for free' if the new processor is faster than the old one and the translation is reasonable.)

This problem can be formulated in accordance with the reference model for a suitable choice of environment and system. Here the system can be chosen somewhat narrowly to include only the translation itself. The concepts of a *well-constructed* program and its *observable behavior* are not available to the translation program. They are environment-controlled and hidden from the translation system. What is visible to the translation program is the syntax for assembly language programs for each of the processors involved. The only value controlled by the system that is visible to the world is the translation itself. System controlled, environment hidden concepts would include programming constructs for the programming language in which the translation program is written. Over each assembly language, we can define an evaluation relation describing how the execution of a program transforms the state of the processor on which it is run. We can also define what it is for a program to be *typable*, what it is for an initial state to be *well-formed*, and what the *visible* components of a state of a processor are. Domain knowledge then tells us that all reasonably well-constructed programs are typable, and links the observable behavior of a program to the visible components of the result of its evaluation in a well-formed initial state.

The specification of this system (the translation program) may be given at a variety of levels. A high-level specification may simply indicate that if we evaluate a program in a well-formed initial state for the old chip, the visible components of the final state are the same as if we evaluate the translation of the program in a comparable well-formed initial state on the new chip. A lower-level specification could provide a translation of instructions on the old processor into those on the new processor. The high-level specification is valuable because it leaves maximum flexibility for the programmer to write the translation program, for example allowing him to optimize the output code to take best advantage of the new architecture. It has the disadvantage, however, of not giving much information about how to write an implementation. On the other hand, the lower-level specification tells the programmer precisely what program to write, providing very little freedom. Such a low-level specification will disallow future versions that take advantage of improved optimization techniques. To get the best of both worlds, it is possible to give specifications of both kinds and then prove that the low level specification is a refinement (in the sense of 9) of the high level specification. Since this notion of refinement is transitive, if we further refine the low-level specification to an implementation, then the implementation will also be a refinement of the high level specification.

7.2 DFC: Distributed Feature Composition

It has proven extremely difficult to find satisfactory specification techniques for telecommunications systems [18]. The behavior of such a system is defined by an ever-expanding set of features, and these features interact extensively. No one has been able to specify feature sets in a way that is sufficiently modular, complete, consistent, and comprehensible for use on a practical scale.

As an alternative to attacking a seemingly unsolvable problem, it may be more productive to try a different problem. The DFC architecture [11] attempts to provide a way to describe feature sets that is modular, complete, comprehensible, and can be checked for consistency and other desirable properties. Its disadvantage from a formal perspective is that these descriptions are not pure specifications—they are architectural descriptions including many phenomena and behaviors that are internal to the telecommunications system.

The primary characteristic of the DFC architecture is that each feature is implemented by one or two component types, and each external call is processed by a dynamically assembled configuration of components and featureless, two-port internal calls. The resulting configuration is analogous to an assembly of pipes and filters, and has the typical advantages of the pipe-and-filter architectural style: feature components are independent, they do not share state, they do not know or depend on which other feature components are at the other ends of their calls (pipes), they behave compositionally, and the set of them is easily enhanced [16].

In a complete formal treatment of the DFC architecture, theory M describes the DFC virtual machine, which can support any feature set. It has the capabilities to run feature-component programs, make featureless internal calls, perform the special DFC routing algorithm (which determines how components are assembled), and provide storage/retrieval functions for restricted global data. When a DFC feature set is implemented as a telecommunications system, this virtual machine is mapped onto a physical telecommunications network; fortunately the virtual machine is well-adapted to doing this efficiently.

In a complete formal treatment of the DFC architecture, theory P describes the feature set and customers, and acts as a program for a DFC virtual machine. P includes feature-component programs and the data used by the routing algorithm. Some of the routing data is customer data; it is constrained by the feature set, but it actually comes from a process called 'provisioning' through which customer data is entered into a telecommunications system.

Theory W must describe the protocols used on the lines and trunks through which the telecommunications system is connected to its environment. It may also include information of a wider scope, such as properties of other telecommunications systems with which this one interoperates.

The requirements R for a telecommunications system should concern the behavior that customers can observe through telephones and other communications devices. There are two practical reasons why R cannot be complete. First, consider telephones that are connected directly to the system through dedicated local lines. Since these telephones are very close to the boundary of the system, any complete description of their behavior would be very similar to a complete specification, which we do not have. Second, consider telephones that are connected to this system only through another telecommunications system, as residential telephones are typically connected to a long-distance network through a local network. Since these telephones are very far from the boundary of our system, our system has very incomplete and indirect control over how they behave.

Thus R is not a complete set of requirements, but rather a set of generally desirable properties that act as partial requirements and a 'sanity check' on everything else. One of the most useful kinds of requirement is a 'non-interference' property derived from a description of an important feature. It states that a particularly crucial property of this feature is unconditionally true for the system as a whole, and is never interfered with by less important features. For example, a customer who calls a 'free' number should never incur charges of any kind when doing so.

Normally, to prove such a property from R, we would invoke the theories W and S. In this case there is no S, so its place must be taken by the conjunction of P and M. which normally imply S rather than substituting for it.

8 Analysis

Relative consistency (Formula 5) is a non-trivial proof obligation and is known to have some drawbacks in modeling. The property is sometimes called the 'input enabled' assumption because it asserts that *any* collection of environment actions permitted by the domain assumptions must be dealt with by the system. What happens if the system is able to perform an action that prevents an environment event from happening, and this capability falls within the scope of the specification (and programming) rather than the domain knowledge? In this case Formula 5 may be too strong. To see this concretely, let us consider the zoo example studied in [10]. We set up the example, describe the problem, and then consider possible approaches to its solution.

The aim is to write a program for controlling the turnstile of a zoo to satisfy the requirement that the number of people who enter does not exceed the number of payments received. The designated terminology shared between the environment and system consists of five terms. Three of these are controlled by the environment:

- push(e): at event e, a visitor pushes the turnstile through a rotation,
- coin(e): at event e, a coin is received, and
- enter(e): at event e, a visitor moves past the turnstile to enjoy the zoo.

The first and second of these are visible to the system, while the third is not. There are two designated terms controlled by the system:

- lock(e): at event e, the system locks the turnstile, and
- unlock(e): at event e, the system unlocks the turnstile.

Thus eh is enter and ev is push, pay and sv is lock, unlock. The system is modeled by an ordering of these kinds of events. The domain knowledge W says that it is impossible to push if the turnstile is locked. Various different specifications are able to meet the requirement. The obvious one is to alternate payments with pushes, but it is also possible to allow a sequence of payments followed by a sequence of pushes. Some approaches, like allowing a push followed by a payment, would allow the environment to perform events that would violate the requirements (eg. there is no domain assumption about the honesty of the visitors).

Let us assume that S is the alternating admission specification: a push is allowed only if preceded by a payment. Although this seems like a perfectly reasonable solution, it is not hard to see that it is not relatively consistent in the sense of Formula 5! To see why, suppose that e is the sequence of events in which visitors push without paying. There *is* a choice of s that is consistent with this: it unlocks the turnstile and never locks it again. Thus we satisfy the hypothesis $\exists s.\ W$. However, the conclusion $\exists s.\ W \wedge S$ is not satisfied for this choice of e: it is not considered an acceptable sequence of events according to the requirements R. Nothing about Formula 5 takes into account the possibility that there may be sequences of environment-controlled events that are consistent with *some* system behavior, but not with the behavior that has been specified.

The problem here can be viewed in two ways: the reference model property of relative consistency is too strong, or we have not modeled this particular problem properly. To repair the example is possible: control needs to be changed so that the system controls pushing. Indeed, this problem can always be addressed by dividing shared phenomena so that the environment performs a 'request' event and the system responds with a 'permit' event. Here the push designation shifts from environment-controlled to system-controlled.

This strategy seems unnatural to us; it complicates the applications in order to simplify the description of the reference model obligations. In the zoo example the result is quite displeasing: the locking and unlocking of the gate seem to become irrelevant since the system apparently has direct control of whether an entry occurs. The thing over which the system really does have control—the ability to lock and unlock—has been suppressed to a lower level in favor of a request/permit abstraction. We are investigating approaches to weakening relative consistency to allow the possibility that environment events are predicated on properties of prior environment and system events. This approach is only necessary for reactive systems, since there is no problem with data processing applications such as *wrap* and DSP. Moreover, there are clearly reactive systems like VTS and the numerous significant systems modeled using the Functional Documentation model [17, 15] that can use the stronger criterion. Details of our progress on an alternative are beyond the scope of this paper.

9 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 such as the one we carried out for VTS. 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, and Insup Lee, 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. To appear in the Conference on Computer Aided Verification, 1998.

[4] Karthikeyan Bhargavan, Carl A. Gunter, Elsa L. Gunter, Michael Jackson, Davor Obradovic, and Pamela Zave. The village telephone system: A case study in formal software engineering. `www.cis.upenn.edu/ ~hol/ vts.ps`, March 1998.

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

[6] C. C. Chang and H. J. Keisler. *Model Theory*, volume 73 of *Studies in Logica and the Foundations of Mathematics.* North-Holland, 1973.

[7] Michael J.C. Gordon and Tom F. Melham. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press, 1993.

[8] M.J.C. Gordon. HOL – A Proof Generating System for Higher-Order Logic. In *Proc. Hardware Verification Workshop, Calgary, Canada*, 1987.

[9] 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.

[10] 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.

[11] Michael Jackson and Pamela Zave. Distributed feature composition: A virtual architecture for telecommunications services. Submitted for publication, September 1997.

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

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

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

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

[16] Mary Shaw and David Garlan. *Software architecture: Perspectives on an emerging discipline.* Prentice-Hall, 1996.

[17] 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.

[18] Hugo Velthuijsen. Issues of non-monotonicity in feature-interaction detection. In K. E. Cheng and T. Ohta, editors, *Feature Interactions in Telecommunications III*, pages 31–42. IOS Press, 1995.

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