In Proc TPHOLS'98, LNCS 1479, pp49-66

THE VILLAGE TELEPHONE SYSTEM: A Case Study in Formal Software Engineering

Karthikeyan Bhargavan[1], Carl A. Gunter[1], Elsa L. Gunter[2], Michael Jackson[3], Davor Obradovic[1], and Pamela Zave *[3]

[1] University of Pennsylvania
[2] Bell Labs, Lucent Technologies
[3] AT&T Laboratories

Abstract. In this paper we illustrate the use of formal methods in the development of a benchmark application we call the *Village Telephone System* which is characteristic of a class of network and telecommunication protocols. The aim is to show an effective integration of methodology and tools in a software engineering task that proceeds from user-level requirements to an implementation. In particular, we employ a general methodology which we advocate for requirements capture and refinement based on a treatment of designated terminology, domain knowledge, requirements, specifications, and implementation. We show how a general-purpose theorem prover (HOL) can provide formal support for all of these components and how a model checker (Mocha) can provide formal support for the specifications and implementation. We develop a new HOL theory of inductive sequences that is suited to modelling reactive systems and provides a common basis for interoperability between HOL and Mocha.

1 Introduction

One of the key problems in the practical adoption of formal methods is that many are usable only at certain stages of the software engineering process and must work with a specific form of data. At AT&T and Bell Labs we have seen a number of instances where formal tools *might have been* useful in a project *if* the kinds of specifications on which such tools work were available. Unfortunately projects do not typically use formal methods in the development of specifications, so the information on which a formal method might be employed is unavailable, or would be very expensive to obtain. Indeed, once a project has chosen not to use formal language early in the development of requirements and specifications for software, it is difficult (or often impossible for all practical purposes) to introduce such formality at a later stage. This suggests that it is essential to find ways in which formal language can be introduced at *early* stages of requirements

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

capture, and there must be effective refinement principles for moving from user-level requirements to an implementation.

Several approaches must be brought together to address this problem effectively. In this paper we explore the problems of modelling and tool integration on an illustrative problem we call the Village Telephone System (VTS). The VTS provides an accessible but non-trivial application similar to many others in the telecommunications and networking domains. We analyze it using a methodology we have developed in [5–7, 4] with formal support provided by the HOL90 general-purpose theorem prover and the Mocha model checker. First we provide a brief overview of the methodology, referring the reader to the cited work for more details. The main body of the paper is devoted to the treatment of the VTS using this methodology.

Principles of good requirements engineering [7] demand that we identify the primitive vocabulary that is available to describe the application domain, and that we provide a precise (albeit informal) explanation of the real-world meaning of each primitive term. The principles also demand that we separate logical assertions into two distinct moods. Assertions made in the indicative mood describe the environment as it would be regardless of the system—they represent domain knowledge. Assertions made in the optative mood describe the environment as we would like it to be because of the system—they represent requirements. A requirement is not necessarily directly implementable by a computer system [6]; it may involve concepts that are not directly visible to the implementor. If it is not, then it must be refined into an implementable specification, using domain knowledge as a resource. Formal requirements engineering culminates in an argument that the specifications and the domain knowledge are consistent, and in a proof that the domain knowledge and the specifications together entail the satisfaction of all the requirements. Similarly, on the development side, the aim is to ensure that the programming of the machine satisfies the specifications.

Our treatment of the Village Telephone System is accordingly organized into a collection of parts, each having a different significance:

Mathematical Foundations provide concepts not already available in existing libraries that are needed for the VTS.

Designated Terminology provides terms to describe the application domain (environment, world), and an informal explanation of their meaning in the real world.

Requirements indicate what the villagers need from their telephone system, described in terms of the designations.

Domain Knowledge provides presumed facts about 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 machine to satisfy the requirements and specifications.

If we got the specifications right, then it will be possible to combine our (presumably correct) domain knowledge about the environment with the specifications

of our system and show that the villagers will have the kind of telephone service they require.

The last five parts in the list above can be grouped into categories of environment and system to emphasize their roles, with the specifications acting as an intermediary between the system and its environment. The "Five Theory Model" can be illustrated as follows:

Generally speaking, the proof obligations are to show that the theories in question are consistent and that, under appropriate assumptions:

- the domain knowledge W, supplemented by the specifications S, satisfies the requirements R, and
- the programming platform M, with its programming P, implements the specifications.

The precise statement of these obligations in Higher-Order Logic is given in the section on designations below (Table 1) because it depends crucially on distinguishing variables (representing events and state) that are controlled by the environment (like a person taking a telephone off-hook) from those that are controlled by the system (like causing a telephone to ring). Details about our refinement principles can be found in [4].

Turning now to our benchmark problem, we shall illustrate our approach for a simple telephone service suited to the needs of a very friendly village. The telephones are fairly conventional: they have a microphone (or mouthpiece) and a speaker (or earpiece) and they ring to alert an incoming call. Taking the phone off-hook when it is ringing answers the incoming call. Putting it back on-hook terminates calls. Taking it off-hook when it is not ringing indicates a desire to make a call. They are less conventional in two respects. First, they have no dialing device, nor is there an operator in the telephone exchange: the exchange is entirely automatic. The maker of a call cannot therefore choose which number to call. The system makes the choice. This is acceptable because the villagers know each other and each other's business so well that a villager wanting to make a call is equally happy to talk to any fellow villager. The second difference is that a villager whose partner in a call has just hung up need only wait, keeping his own phone off-hook, and the system will immediately try to find him another conversation partner. A variant of this system we have investigated, but will not consider in this paper, has such phones enter a "drooping" state where they cannot be connected until they go back on-hook. There can, of course, be no guarantee that the system will always find a partner for every villager who wants to talk, because a ringing phone may be left unanswered indefinitely, and there may be no-one left available to be rung. However, we are guaranteed that the system will ring someone if this is possible.

Although VTS is not in itself a product of any telephone company, it is a fairly typical protocol resembling communication services such as anycast or

chat lines. There is a variety of possible implementations representing trade-offs such as the likelihood of finding a partner and other factors. At one extreme (broadcast) an off-hook event could cause all on-hook telephones to alert and, at another extreme (hotlines), each telephone t could have its own pre-determined unique partner which alerts in response to an off-hook event of t. An intermediate solution (anycast) could cause an undetermined on-hook telephone to alert in response to an off-hook event. Each of these approaches has various refinements, such as allowing any off-hook event to make a connection to an existing off-hook telephone that has not yet received a connection. The VTS is therefore more interesting than its cousin, Plain Old Telephone Service (POTS), in which a uniquely designated telephone is alerted as a result of dialing an off-hook telephone. Between the two lie a range of interesting services in which an off-hook telephone seeks a connection with any of a specified collection of on-hook telephones. VTS represents the extreme in which every such telephone is a candidate for connection, while POTS represents the extreme in which only one other telephone is a candidate. In this middle ground fall services such as 800 numbers in the North American system where an incoming call is assigned to one of a group of operators, possibly with queueing if all operators are engaged.

We have used several systems in the development of the VTS: the HOL90 theorem prover, the Mocha model checker, and the SML programming language. This diversity was intended to help us explore the parts of the task best treated by each tool. We have allowed some overlap in order to carry out comparisons, but have also used the tools in exploring distinct solutions and in different parts of the development. In particular, HOL has been used for all phases of the development except the programming, whereas SML is used only for the programming. Mocha is used to provide a specification and also programming. Our SML implementation uses an anycast solution together with what we call the "greedy" connection rule while we considered a broadcast solution in the Mocha specification and implementation.

The paper is divided into sections representing each of the parts we discussed earlier for our methodology (the program and programming platform are combined in a single implementation section). Each section emphasizes what we view as the most interesting themes. For instance, the mathematical foundations section describes an HOL model that we have tuned for use on reactive systems like the VTS, and the implementation section considers the challenge of bridging between formal specification language and executable programs. Another issue is the set of tradeoffs involved in using a general-purpose system (HOL90) versus a special-purpose one (Mocha). The final section provides some conclusions.

2 Mathematical Foundations

As is usual with projects in HOL, we found it desirable to build up a body of fairly general purpose mathematics as a foundation of the requirements and specification of the village telephone system in HOL. This background should be useful for the description of reactive systems in general. There is a basic temporal

theory given by inductive sequences, and a theory of finite state machines with a specialization to toggles.

2.1 Inductive Sequences

The formalization of reactive systems has typically been founded on some notion of sequences of events. The paper [3] discusses differences in four theorem prover formalizations of possibly infinite sequences. We have chosen to treat sequences in a way that differs from these in two fundamental ways. The approaches taken so far have been *explicit* in that they build a specific model for sequences and then derive properties. Here, we shall be taking an *implicit*, or axiomatic approach. The definition of a sequence is given by:

$$\begin{array}{l}\forall order\ \ domain.\ \ \mathsf{inductive\_sequence}\ (order, domain)\ \ = \\ \quad \mathsf{transitive}\ (order, domain)\ \ \wedge\ \ \mathsf{irreflexive}\ (order, domain)\ \wedge \\ \quad \mathsf{nondense}\ (order, domain)\ \ \wedge\ \ (\exists\ f.\ \mathsf{least}\ (order, domain)\ \ f)\ \wedge \\ \quad (\forall Inv.\ \ ((\forall f.\ \mathsf{first}\ (order, domain)\ \ f\ \ \Rightarrow\ \ Inv(f))\ \wedge \\ \qquad\qquad (\forall i\ \ j.\ \mathsf{successor}(order, domain)\ \ (i, j)\ \wedge\ Inv(i)\ \ \Rightarrow\ \ Inv(j)))\ \ \Rightarrow \\ \qquad \forall i.\ \ domain(i)\ \ \Rightarrow\ \ Inv(i))\end{array}$$

An ordering is non-dense if every element that is not the first has an immediate predecessor, and if it has anything greater than it, then it has an immediate successor. The last part is the principle of induction. Any set and ordering that is an inductive sequence is isomorphic to an initial segment of the natural numbers. Still, by not restricting ourselves to that particular model, we get certain properties practically for free. For example, we automatically get that any non-empty subset of an inductive sequence is again an inductive sequence.

Using inductive sequences, we can now develop a rich temporal theory appropriate for reasoning about reactive systems. For example, given a predicate P on events of an inductive sequence, we can define predicates like $\mathsf{previously}(P)$, which says that P holds of the previous event, and $\mathsf{throughout}(P)$ which says that P holds continuously throughout some interval. We have one-step induction (which is more conveniently used in conjunction with $\mathsf{previously}$) and general induction. Thus, the same machinery that is available to the explicit versions of sequences is available for the implicit one.

The previous approaches also are based, directly or indirectly, on mappings from some ordered set to actions, telling what action occurred at a given time. We have taken the dual approach. Actions are represented as predicates stating at which events they occur. Thus $\mathsf{on}(t)$ is a predicate on events that indicates all those times when the telephone t went on-hook. If we fix the set of action predicates, then we can recreate the inverse mapping from events to actions. However, by doing the mapping this way around, we can more easily extend our system to include more actions and readily compose two systems in parallel. Also, to express a system with true concurrency requires no extra effort, while if the mapping is done the reverse of our way, then true concurrency requires switching from sequences of actions to sequences of sets of actions. On the whole, we believe

this formalization of sequences of actions will prove to carry less overhead for many applications than previous methods.

2.2 Finite state machines

Finite state machines (FSM's) are one of the most commonly used specification formalisms. A variety of descriptive techniques are based on FSM's, different techniques incorporating them in different ways. For example, we could be interested only in the sequence of *states* traversed, or only in the sequence of *transitions* taken. Our objective is to develop a reasonably general HOL model for FSM's that could handle different variations as special cases.

Finite state machines recognize inductive sequences. Formally, fsm is a predicate defined over the 5-tupples $(v, q, q0, f, r)$, where v is a predicate that provides a vocabulary of actions:

$v : (event \to \mathsf{bool}) \to \mathsf{bool}$	the set of transition *labels*
$q : state \to \mathsf{bool}$	the set of *states*
$q0 : state \to \mathsf{bool}$	the set of *initial states*
$f : state \to \mathsf{bool}$	the set of *final states*
$r : state \times (event \to \mathsf{bool}) \times state \to \mathsf{bool}$	the set of *transitions*.

Notice that transitions are labeled with predicates whose role is to determine the availability of the transition at any given moment.

$$\begin{array}{l}\mathsf{fsm}\ (v,q,q0,f,r)\ =\ (\mathsf{finite}\ q)\ \wedge\ (q0\ \subseteq\ q)\ \wedge\ (f\ \subseteq\ q)\ \wedge \\ \quad (\forall\ l\ s1\ s2.\ ((s1,l,s2)\ \in\ r)\ \Rightarrow\ (s1\ \in\ q\ \wedge\ s2\ \in\ q\ \wedge\ l\ \in\ v))\ \wedge \\ \quad (\mathsf{pairwise\_disjoint}\ \{l\ \mid\ \exists\ s1\ s2.\ (s1,l,s2)\ \in\ r\})\end{array}$$

This models nondeterministic FSM's. We need to define the way in which an FSM interprets an inductive sequence. For that purpose, we define the relation pstates $e\ s$ that says when a state s can be entered after an event e. The relation is defined by rule induction:

$$\frac{\mathsf{least}\ e \quad s \in q0}{\mathsf{pstates}\ e\ s}\quad (\mathit{Init})$$

$$\frac{\mathsf{pstates}\ e1\ s \quad \mathsf{successor}\ (e1,e2) \quad (s,l,t) \in r \quad e2 \in l}{\mathsf{pstates}\ e2\ t}\quad (\mathit{Step})$$

$$\frac{\mathsf{pstates}\ e1\ s \quad \mathsf{successor}\ (e1,e2) \quad \forall l \in v.\ e2 \notin l}{\mathsf{pstates}\ e2\ s}\quad (\mathit{Stay}).$$

The *Stay* rule says that all the events which the machine does not mention at all are ignored (filtered out).

An inductive sequence is accepted by an FSM if, at every point, it is guaranteed the ability to reach a final state. Formally,

$$\begin{array}{l}\mathsf{accept}\ (v,q,q0,f,r)\ (order,domain)\ = \\ \quad \forall\ e1.\ (e1\ \in\ domain\ \Rightarrow \\ \quad\quad \exists\ e2\ s.\ (s\ \in\ f)\ \wedge\ ((e1 = e2)\ \vee\ order(e1,e2))\ \wedge\ \mathsf{pstates}\ e2\ s).\end{array}$$

This definition takes care of both finite and infinite inductive sequences. In the finite case it coincides with the classical definition of acceptance by ending up in a final state. In the infinite case it coincides with acceptance by Buchi automata.

We also defined deterministic FSM's as a special case and proved some basic results about them. An interesting class of deterministic FSM's are *toggle FSM's*. A toggle FSM is an FSM with exactly two states and two disjointly labeled transitions between them. Given two sets of events *go_on* and *go_off* they determine a toggle FSM iff:

$$\begin{array}{l}\mathsf{toggle}\ (go\_on, go\_off) \ = \ \mathsf{fsm}\ (\mathsf{toggle\_fsm}(go\_on, go\_off) \ \wedge \\ \quad \mathsf{accept}\ (\mathsf{toggle\_fsm}(go\_on, go\_off))\ (order, domain)\end{array}$$

where

$$\begin{array}{l}\mathsf{toggle\_fsm}(go\_on, go\_off) \ = \\ \quad (\{go\_on; go\_off\}, \{0; 1\}, \{0\}, \{0; 1\}, \{(0, \mathsf{go\_on}, 1), (1, \mathsf{go\_off}, 0)\end{array}$$

3 Designated Terminology

This part of the VTS description presents the primitive vocabulary that is available for use to describe the application domain (environment, world). It also explains the real-world meaning of each primitive term. Obviously these explanations are informal; if they were formal, then the terms would not be primitive. In general, designated terminology must be classified into one of four categories according to control and visibility: environment-controlled, system-hidden; environment-controlled, system-visible; system-controlled, environment-visible; and system-controlled, environment-hidden. When we need to represent these variables in mathematical formulae, we shall write them as *eh*, *ev*, *sv*, and *sh*, where each of these is to be viewed as a list of variables. The system-controlled and environment-hidden variables *sh*, only arise within the implementation and will not be covered here. The purpose of the designations is to clarify the role these terms may play in the formation of the domain knowledge, specification and requirements. It also is critical in formulating the basic theorems that need to relate these components. Using the variable classification, we can represent the domain knowledge by $\mathsf{W}(eh, ev, sv)$, represent the requirements by $\mathsf{R}(eh, ev, sv)$, represent the specification by $\mathsf{S}(ev, sv)$, represent (as an input-output relation) a program implementing the specification by $\mathsf{P}(ev, sv, sh)$ and represent knowledge of the programming platform (machine) on which the program will be run by $\mathsf{M}(ev, sv, sh)$. Notice that the domain knowledge and the requirements cannot reference those variables controlled by the system and hidden from the environment, and that the specification can only reference those variables visible to both the system and the environment. Suppressing the arguments, the ultimate theorems we wish to hold are given in Table 1. Formulas (1) and (2) are consistency properties and (3) is the correctness of the implementation relative to the domain knowledge and requirements. From (2) and (3), we can prove the consistency of the requirements relative to the domain knowledge. To prove (2) and (3), we will factor through the specification. If we prove (4), (5) and (6) then we can derive both (2) and (3) from them. A major part of the point of

Table 1. Proof Obligations for Refinements

$\exists\ eh\ ev\ sv.\mathsf{W}$	(1)
$\forall eh\ ev.(\exists\ sv.\mathsf{W}) \Rightarrow (\exists\ sv.\mathsf{W} \wedge \mathsf{M} \wedge \mathsf{P})$	(2)
$\forall eh\ ev\ sv.\mathsf{W} \wedge \mathsf{M} \wedge \mathsf{P} \Rightarrow \mathsf{R}$	(3)
$\forall eh\ ev\ sv.\mathsf{W} \wedge \mathsf{S} \Rightarrow \mathsf{R}$	(4)
$\forall eh\ ev.(\exists\ sv.\mathsf{W}) \Rightarrow (\exists\ sv.\mathsf{S}) \wedge (\forall sv.\mathsf{S} \Rightarrow \mathsf{W})$	(5)
$\forall ev.(\exists\ sv.\mathsf{S}) \Rightarrow (\exists\ sv\ sh.\mathsf{M} \wedge \mathsf{P}) \wedge (\forall sv\ sh.(\mathsf{M} \wedge \mathsf{P}) \Rightarrow \mathsf{S})$	(6)

this factorization is that on the one hand, the person writing the specification need only worry about satisfying (1), (4) and (5) without any concern for the particulars of any program that might implement it, while the person writing the program need only worry about satisfying (5) without any knowledge of the domian knowledge or the original requirements. Formula (5) for the specification is a bit stronger than the corresponding formula (2) for the program and programming platform. Formula (5) asserts that for all values from the environment that do not contradict the domain knowledge, the specification relates some value from the system, and all such values from the system must satisfy the domain knowledge. It turns out that the correspondingly stronger version of formula (2) also follows from (4), (5), and (6).

The designated terminology describing time, people, telephones, sounds, their actions and interactions is as follows:

- **Environment-controlled, system-visible:**
  - $\mathsf{event}(E)$: E is an atomic event.
  - $\mathsf{earlier}(E1, E2)$: event $E1$ is earlier than event $E2$, and
  - $\mathsf{tel}(t)$: t is a telephone in the village;
  - $\mathsf{on}(t)(E)$: E is an event where telephone t goes onhook.
  - $\mathsf{off}(t)(E)$: E is an event in which telephone t goes offhook.
- **Environment-controlled, system-hidden:**
  - $\mathsf{person}(p)$: p is person in the village;
  - $\mathsf{sound}(s)$: s is a unit instance (or packet) of sound.
  - $\mathsf{go\_near\_phone}(p, t)(E)$: a person p goes near (enough to be heard over) a telephone t at an event E.
  - $\mathsf{go\_away\_from\_phone}(p, t)(E)$: a person p goes away (enough not to be heard over) a telephone t at an event E.
  - $\mathsf{make\_sound}(p, s)(E)$: a person p makes a sound s at an event E.
  - $\mathsf{hear\_sound}(p, s)(E)$: a person p hears the sound s at an event E.
  - $\mathsf{transmit}(t_1, t_2, s)(E)$: the sound s is transmitted from telephone t_1 to telephone t_2 at an event E.
- **System-controlled, environment-visible:**
  - $\mathsf{then\_alerting}(t)(E)$: Immediately after event E, telephone t is in an alerting state (that is, the telephone is 'ringing').

- then_connected$(t1, t2)(E)$: Immediately after event E, telephones $t1$ and $t2$ have a talking connection.

The predicates tel, person, and sound are 'timeless' facts treated as constants. The other predicates above are partially curried on events to facilitate their use with the general temporal theory. Most of the predicates are just what would be expected. The treatment of sound is a little unusual. Since we are assuming that time is discrete, we assume that sound comes in discrete units as well. We also associate with a sound its origin so that sounds made by different people are different sounds.

The temporal theory assumes that events are instantaneous: the actions of the system (telephone system) are sufficiently fast that users perceive them as happening in no time, for all practical purposes. The state of the telephone system changes only at events, so state predicates are often defined using event boundaries. For instance, a telephone t that satisfies then_alerting$(t)(E)$ is one that began to alert at event E or was alerting prior to E and continued to alert after E. State is often viewed in terms of *immediately before* E (alerting_then) and *immediately after* E (then_alerting). The alerting state immediately before E is a defined predicate:

$$\forall E\ t.\ \text{alerting\_then}\ t\ E\ =\ \text{previously}(\text{then\_alerting}\ t)\ E.$$

Using the designated terminolgy, we have built up a considerable vocabulary of defined terminolgy. One such example is alerting_then. We omit the definitions here, but assume that the names are adequately suggestive to allow the reader to determine what the definitions are.

4 Requirements

Because this is a very friendly village we require the system to make it as easy as possible for villagers to talk to each other. Intuitively, the requirement is that if a villager wants to talk to somebody the system will make an effort to find a suitable partner—that is, another villager who is offhook and not engaged in another conversation, and therefore free to talk. This effort may include alerting one or more villagers whose phones are onhoook in the hope that a phone will then go offhook and can be connected. There are many possible versions of these informal requirements. In all versions we assume that time is discrete (this is stipulated by the temporal theory) and that the system is fast enough to complete its response to each event before the next environment event occurs (this is sometimes called the "reactive system hypothesis"). In our case this means that then_connected and then_alerting can be viewed as instantaneous state changes.

The first thing that anybody would want out of a telephone system is that communication can happen:

PR0(near_phone_then, is_offhook_then, connected_then, make_sound, hear_sound) =

$$\forall E\ p1\ p2\ t1\ t2\ s.\ (\mathsf{near\_phone\_then}(p1, t1)\ E\ \wedge\ \mathsf{is\_offhook\_then}\ t1\ E\ \wedge$$
$$\mathsf{near\_phone\_then}(p2, t2)\ E\ \wedge\ \mathsf{is\_offhook\_then}\ t2\ E\ \wedge$$
$$\mathsf{connected\_then}(t1, t2)\ E\ \wedge\ \mathsf{make\_sound}(p1, s)\ E)\ \Rightarrow\ \mathsf{hear\_sound}(p2, s)\ E$$

In what remains, we will leave the arguments to PRn implicit. Another thing people expect from their phone is a degree of privacy:

$$\mathsf{PR1}\ =\ \forall E\ p\ s\ t1\ t2.(\mathsf{make\_sound}(p, s)\ E\ \wedge\ \mathsf{is\_onhook\_then}\ t1\ E)\ \Rightarrow$$
$$(\neg\mathsf{transmit\_sound}(t1, t2)\ E\ \wedge\ \neg\mathsf{transmit\_sound}(t2, t1)\ E)$$

A bit of politeness is that an offhook telephone should not be alerting:

$$\mathsf{PR2}\ =\ \forall E\ t.\ \mathsf{then\_offhook}\ t\ E\ \Rightarrow\ \neg\mathsf{then\_alerting}\ t\ E$$

Connections are reliable in the sense that a connection is not broken (or even replaced by another connection) until one of its participants goes onhook:

$$\mathsf{PR3}\ =\ \forall E\ t1\ t2.$$
$$(\mathsf{connected\_then}\ (t1, t2)\ E\ \wedge\ \neg\mathsf{then\_connected}\ (t1, t2)\ E)$$
$$\Rightarrow\ (\mathsf{on}\ t1\ E\ \vee\ \mathsf{on}\ t2\ E)$$

Alerting is also reliable in the sense that an answered phone (one that goes offhook while it is alerting) immediately enters the talking state (that is, it is connected to some other phone). That other phone may of course go onhook in the very next event, but between the two events the answered phone is in the talking state:

$$\mathsf{PR4}\ =\ \forall E\ t.\ \mathsf{answer}\ t\ E\ \Rightarrow\ \mathsf{then\_talking}\ t\ E$$

When somebody is requesting a connection (by having taken their phone offhook when it was not alerting and not having been connected or having hung up yet) and there is an onhook phone, then some phone is alerting.

$$\mathsf{PR5}\ =\ \forall E\ t1.\ \mathsf{then\_requesting}\ t1\ E\ \wedge\ (\exists\ t2.\ \mathsf{then\_onhook}\ t2\ E) \Rightarrow$$
$$(\exists\ t3.\ \mathsf{then\_alerting}\ t3\ E)$$

The partial requirements PR1 through PR5 are basic to any version of this telephone service, and we have included them in each of the speculative set of requirements we investigated. Let us therefore refer to the following formula as partial requirement B:

$$\mathsf{B}\ =\ \mathsf{PR0}\ \wedge\ \mathsf{PR1}\ \wedge\ \mathsf{PR2}\ \wedge\ \mathsf{PR3}\ \wedge\ \mathsf{PR4}\ \wedge\ \mathsf{PR5}$$

In addition to the basic requirements, there are two alternative ways of handling phones that loose a connection. A telephone is said to *droop* if it was connected to another phone which hangs up, and remains drooping until it either hangs up or is connected to another phone. There are two evident options for how to handle a drooping phone. One option is to treat a drooping phone the same as a requesting phone. In which case we have a requirement for drooping phones that is the same as PR5 for requesting phones.

$$\mathsf{PR6} \;=\; \forall E\ t1.\ \mathsf{then\_drooping}\ t1\ E\ \wedge\ (\exists\ t2.\ \mathsf{then\_onhook}\ t2\ E) \Rightarrow (\exists\ t3.\ \mathsf{then\_alerting}\ t3\ E)$$

The other option (which is the one used by POTS) is to treat it as unavailable until it goes onhook. In this case we would have the requirement:

$$\mathsf{PR6'} \;=\; \forall E\ t.\ (\mathsf{droop}\ t\ E\ \vee\ \mathsf{drooping\_then}\ t\ E) \;\Rightarrow\; \neg\mathsf{then\_talking}\ t\ E$$

To cover the option where drooping phones are treated the same as requesting phones, we will say that a phone is asking if it is either requesting or drooping. There is one last requirement that we have for the system, namely that it be fair to the callers by treating them on a first come, first served basis. Assuming drooping telephones are handled the same as requesting phones, this yields:

$$\begin{aligned}\mathsf{PR7} \;=\; & \forall E1\ t1\ t2.(\mathsf{asking\_then}\ t1\ F1\ \wedge \\ & (\mathsf{ask}\ t2\ F1\ \vee \\ & (\mathsf{asking\_then}\ t2\ F1\ \wedge \\ & \exists E2.\ \mathsf{throughout}\ (\mathsf{asking\_then}\ t1)\ E2\ F1\ \wedge \\ & \neg\mathsf{throughout}\ (\mathsf{asking\_then}\ t2)\ E2\ F1))) \;\Rightarrow \\ & (\mathsf{then\_talking}\ t2\ F1 \;\Rightarrow\; \mathsf{then\_talking}\ t1\ F1)\end{aligned}$$

For the case where drooping telephones are treated as unavailable, PR7' is derived from PR7 by replacing all occurrences of asking_then by requesting_then. In the specifications given later in this paper we have focused on the case where drooping phones are treated the same as requesting phones. Therefore our requirements are

$$\mathsf{R} \;=\; \mathsf{B}\ \wedge\ \mathsf{PR6}\ \wedge\ \mathsf{PR7}$$

We see no inherent difficulty with deriving specifications for the alternate system which treats drooping phones as unavailable.

Note that there is a great deal of non-determinism in our requirements. For example, none of our requirements directly stipulates a choice of caller-callee pairings. Nor do we stipulate that there should, or should not, ever be more than one phone alerting. Some of this non-determism will be restricted by the choice of specification, but much will be passed on for the program to decide.

5 Domain Knowledge

Our domain knowledge for the VTS is a collection of facts about the environment as we choose to model it for the purposes of our system. First of all, we assert that types of arguments in some of our predicates are as expected:

$$\begin{aligned}\mathsf{K0} \;=\; & (\forall E1\ E2.\ \mathsf{earlier}(E1, E2) \;\Rightarrow\; \mathsf{event}\ E1\ \wedge\ \mathsf{event}\ E2)\ \wedge \\ & (\forall t\ E.\ \mathsf{on}\ t\ E \;\Rightarrow\; \mathsf{tel}\ t\ \wedge\ \mathsf{event}\ E)\ \wedge \quad \cdots \quad \wedge \\ & (\forall t1\ t2\ E.\ \mathsf{then\_connected}\ (t1, t2)\ E \;\Rightarrow \\ & \quad \mathsf{tel}\ t1\ \wedge\ \mathsf{tel}\ t2\ \wedge\ \mathsf{event}\ E)\end{aligned}$$

More significantly, the designated relation earlier is a nondense total order over 'events', there is an initial event, and an induction principle. Using the mathematical foundations we can state this as

$$\mathsf{K1} \;=\; \mathsf{inductive\_sequence}(\mathsf{earlier},\ \mathsf{event})$$

Additionally, off and on events are in disjoint classes and are not initial events. At any telephone, off and on events alternate strictly, beginning with an off event so we wish to model telephones as toggles:

$$\mathsf{K2} \;=\; \forall\ t.\ \mathsf{tel}\ t \;\Rightarrow\; \mathsf{toggle}(\mathsf{off}\ t,\ \mathsf{on}\ t)$$

(The constant toggle actually takes (earlier, event) as an additional argument, but we have omitted it here for the sake of conciseness.)

At most one telephone is going offhook or onhook at any given time:

$$\mathsf{K3} \;=\; \forall E\ t1\ t2.\ (\mathsf{off}\ t1\ E\ \vee\ \mathsf{on}\ t1\ E)\ \wedge\ (\mathsf{off}\ t2\ E\ \vee\ \mathsf{on}\ t2\ E) \Rightarrow (t1\ =\ t2)$$

Another expectation is that then_connected is an irreflexive relation, that is, no telephone is ever connected to itself, and connections are symmetric (perhaps because of the hardware that has been previously agreed upon).

$$\mathsf{K4} \;=\; \forall E\ t.\ \neg\mathsf{then\_connected}\ (t,t)\ E$$
$$\mathsf{K5} \;=\; \forall E\ t1\ t2.\ \mathsf{then\_connected}(t1,t2)\ E\ =\ \mathsf{then\_connected}(t2,t1)\ E$$

Moreover, connections are in pairs:

$$\mathsf{K6} \;=\; \forall E\ t1\ t2\ t3.\ \mathsf{then\_connected}(t1,t2)\ E\ \wedge\ \mathsf{then\_connected}(t1,t3)\ E\ \Rightarrow\ (t2\ =\ t3)$$

Some telephone services have connections called 'conference bridges' that allow three or more parties to be connected, but the village doesn't have this.

The rest of the domain knowledge is about people, sounds and their relation to telephones. First, people are toggles with respect to going near and going away from telephones:

$$\mathsf{K7} \;=\; \forall p\ t.\ \mathsf{person}\ p\ \Rightarrow\ \mathsf{toggle}(\mathsf{go\_near\_phone}\ t,\ \mathsf{go\_away\_from\_phone}\ t)$$

If a person is near a phone $p1$ which is offhook and connected to a phone $p2$ which is also offhook, and the person makes a sound, then that sound is transmitted from $p1$ to $p2$.

$$\mathsf{K8} \;=\; \forall E\ p\ t1\ t2\ s.\ \mathsf{near\_phone\_then}\ (p,t1)\ E\ \wedge\ \mathsf{is\_offhook\_then}\ t1\ E\ \wedge\ \mathsf{connected\_then}\ (t1,t2)\ E\ \wedge\ \mathsf{is\_offhook\_then}\ t2\ E\ \wedge\ \mathsf{make\_sound}\ (p,s)\ E \Rightarrow \mathsf{transmit\_sound}\ (t1,t2,s)\ E$$

If a person is near a phone and a sound is transmitted to that phone, then it is conveyed to that person.[1]

[1] This can be considered analogous to conveying a packet to an application-level program by placing it in a buffer that is accessible by the application. There may be no guarantee that the application will "make use" of the packet, just as VTS makes no guarantee that a person will listen to a sound.

$$\mathsf{K9} \;=\; \forall E\ p\ t\ s.\ \mathsf{near\_phone\_then}\ (p,t)\ E \wedge (\exists\ t1.\ \mathsf{transmit\_sound}\ (t1,t,s)\ E) \;\Rightarrow\; \mathsf{hear\_sound}\ (p,t)\ E$$

Lastly, we assert that if a sound is transmitted from one phone to another, they must be connected.

$$\mathsf{K10} \;=\; \forall E\ t1\ t2\ s.\ \mathsf{transmit\_sound}\ (t1,t2,s)\ E \;\Rightarrow\; \mathsf{connected\_then}(t1,t2)\ E$$

Note that K8 and K9 are sufficient to prove the requirement PR0. It is not true that all the requirements follow from the domain knowledge (PR1 does not, for example), but it is not unreasonable to expect it to happen some of the time.

6 Specifications

The principle attribute of a specification is that it lies in the common vocabulary of the environment and system but still has enough information to entail the requirement, given the domain knowledge. Viewed as a progression from user-level requirements to the development of a machine to satisfy those requirements, it can be viewed as a reduction of the requirements to observable properties of the machine. In the VTS this entails the reduction of requirements that speak of people and sounds to ones that speak of off-hook and on-hook events (which the machine detects but does not control) and telephones in alerting and connected states (which the machine can control). It also may entail reductions in the range of available solutions as it narrows possibilities by stipulating particular approaches. We describe two specifications, the first is done with HOL and uses an anycast solution, while the second is done in Mocha and uses a broadcast solution.

Up to this point, HOL has been our sole platform for formalizing aspects of the VTS. At this point, we are expanding to make use of a second system, Mocha. The question arises, what is the relation between a specicifation given in Mocha and one given in HOL. The answer is that the Mocha specification is directly translatable into HOL because the underlying semantics of time in Mocha, that of a *round*, coincides with that of an event in an inductive sequence in HOL, and the module variables map directly to the designated terminology. We actually studied different specifications with the two different systems, but the Mocha specification is readily expressible in HOL.

6.1 Specifications in HOL

There are essentially two cases because a phone can only do two things: go offhook and go onhook. Within each of these, there are essentially two cases again. Going offhook can happen when the phone is alerting (in which case it is a *answer* event) or it can happen when phone is not alerting (in which case it is a *request* event). Going onhook can happen when the phone is connected (a *disconnect* event), or when it is not (a *withdraw* event). We therefore organize the specification into five formulas: Initial, Answer, Request, Disconnect, and

Withdraw. For each of these we provide an abbreviated English explanation that relies on certain invariants the system satisfies. In developing the specification in HOL, we gave two versions, a "fat" version that included clauses for many cases which (one can prove in retrospect) cannot occur, and the other being a "lean" version which we describe below. We were able to prove that in the presence of domain knowledge the two specifications are the logically equivalent.

The Initial Event: Immediately after the inital event no telephones are alerting or connected.

Answer Events: Assume telephone $t1$ answers at event E. Then there is a phone $t2$ which is asking and which is the unique phone connected to $t1$ after E, and any other pair of phones is connected after E iff it was connected before E. That is, $t1$ connects to some asking phone $t2$, and all standing connections are unaffected. After E the phone $t1$ is no longer alerting and, except for $t1$, a phone is alerting after E iff it was alerting before E.

The "fat" formula for answer events is more complicated because it covers all cases, such as what happens when a phone other than $t2$ is also asking: "if there is another phone besides $t2$ that is asking and $t1$ was the only phone alerting before E, then there is a phone t3 that was onhook before E and starts to alert after E". However, the "fat" set of specification formulae can be used to show that at most one phone can be asking at any one time: the greedy connection rule would connect any pair of simultaneously asking phones. Another point of interest: the "fat" specification implies that there is at most one alerting phone, so, in fact, after E there are *no* alerting phones.

Request Events: Suppose $t1$ is a phone that was not alerting when it goes offhook at E; that is, $t1$ requests a connection at E. If there is another phone $t2$ that is asking, then $t1$ is connected to $t2$, all other standing connections are unaffected, and there are no alerting phones after E. If no other phone is asking, then the standing connections are the same as before E, and some phone that is onhook at E begins to alert, if there is any onhook phone.

Again, this phrasing is based on a variety of invariants like the fact that at most one phone can be alerting at any time and the fact that if no phone is asking then no phone is alerting.

Disconnect Events: Suppose $t1$ goes onhook at event E, where $t1$ was connected to $t2$ immediately before E. Then, after E the connection between $t1$ and $t2$ ends and there are two possibilities for what happens to $t2$. Either another phone was asking and then $t2$ is connected to it and all alerting phones stop alerting, or no other phone was asking, so no new connection is made, and an onhook phone begins to alert. In either case all other connections are unaffected.

Withdraw: Suppose $t1$ goes onhook at event E, but where $t1$ was not connected immediately before E. Then, the set of standing connections is left unaffected, and all phones stop alerting.

In HOL, we have proved a collection of invariants of this specification, such as those mentioned above, and we have proved the equivalence of the two specifications. We have also proved that the specification which covers all cases, including those that cannot arise, satisies the reduction theorem given by formulae (4) and (5) in Table 1. Using the equivalence of the specification under domain knowledge, we then showed the simplified specification also satified formulae (4) and (5). Therefore, either of the specifications may be passed on to developers to build a program to satify it. The developers need not know anything about the original requirements or the domain knowledge; if a program is supplied that satifies formula (6), then we are guaranteed (from a theorem in HOL) that the desired formula (3) will hold.

6.2 Specification in Mocha

Mocha [2] is a model-checking verification system, which uses reactive modules [1] as a modelling language and state invariants for specifications. We specify the VTS using reactive modules.

Reactive Modules A reactive module is a collection of synchronously updated variables. A key concept is that of a round, which is the time-step at which a variable may be updated. There is an initial round when the variables are initialized. Subsequent rounds are called update rounds. The sematics of rounds is the same as the semantics of events in an inductive sequence. Therefore, it is legitimate to identify these two notions, and we shall refer to rounds and events interchangeably from here on.

Formally, a module consists of *external* variables (*ev*), which are inputs to the module; *private* variables (*sh*), which are updated locally but are invisible outside the module; *interface* variables (*sv*), which are updated locally and are visible outside the module. Updated variable values can depend on the current and previous values of any of the variables in the module as long as there are no circular dependencies. Thus each variable is actually a function from rounds to values. The module expresses a set of predicates on the values of the variables in any round.

A specification for the variables in a module can be expressed by writing an invariant for it. An invariant is a condition on the variable values that is expected to hold in all rounds. Very often, the invariant mechanism is not expressive enough for temporal specifications for the module because it does not allow predicates over rounds. In such cases, we use another module to *monitor* the relevant variables. The monitor module sets a flag whenever any condition is violated. Then the specification can be expressed as an invariant of the flag value.

Specifying the VTS In the VTS, the system and the environment are reactive systems which respond to conditions at each event. All the variables are predicated over events. Consequently, we can naturally model the VTS and its specification using reactive modules.

The specification for the VTS is expressed by defining a monitor module with a flag variable which is initially true and goes false if any of the following rules are violated in a round:

Consistency: At any round, no phone is connected to itself, connections are symmetric, and no phone is connected to two phones.

Initial: After the initial round, no telephones are alerting or connected

Answer: At any update round, if a phone is answered, then it stops alerting, there is exactly one asking phone, it is connected to every asking phone, and no standing connections are affected.

Request: At any update round, if a phone requests, then no other phone is asking, and all on-hook phones start alerting.

On-Hook: At any update round, if a talking phone goes on-hook, all its connections are broken; if a talking phone goes on-hook when another phone was asking, the phone's partner gets connected to the asking phone; if a talking phone goes on-hook, all non-alerting phones start alerting; and if a non-talking phone goes on-hook, all alerting phones stop alerting.

The specification is then expressed as an invariant that the flag is always (in all rounds) true.

This specification differs from the HOL specification in exactly one aspect. Here the system is expected to alert all on-hook phones (broadcast) when a connection is requested, as opposed to exactly one on-hook phone (anycast) in the HOL case. We assert that this assumption does not violate the requirements. We have not formally proved that (4) and (5) hold for this specification, but in light of the proof of (4) and (5) for the HOL specification and the close relation between the two specifications, we believe that it can be proved for the reactive module specification as well.

7 Implementations

As was with the case with the specifications, we have given two implementations: one in SML satisfying the simplified HOL specification, and one in Mocha, satisfying the Mocha specification.

7.1 Implementation in SML

The HOL specification reads as a large case statement relating an onhook or offhook event (or initial event) to the set of connections and the set of alerting telephones after the event, given enough history to know which phones are already onhook, offhook, alerting, or connected (and to whom). The SML implementation consists of a state variable containing the current set of connections, a state variable containing the current set of alerting phones, and a recursive program over a stream of onhook and offhook events, yielding state changes to the set of connections and the set of alerting phones. The loop of the recursive

program takes as arguments the current set of onhook phones as well the current event. It generates the changes to the state of the connections and alerting phones, and returns the new set of onhook phones. This internal record of the the set of phones onhook is a machine-controlled, world-hidden variable that mirrors the the defined term onhook_then. The loop mirrors the case statement of the simplified specification very closely. We did not perform a formal proof that the program satisified the specification (*i.e.* formula (6)), but we did do an informal proof. A formal proof should be possible, given the semantics of SML encoded in HOL, but since the similarity of the program and the specification was so great, there seemed diminished value in doing so.

7.2 Implementation in Mocha

In Mocha we implemented VTS as a reactive module with environment controlled variables $\mathsf{on}(t)$, $\mathsf{off}(t)$ as external (input) variables and $\mathsf{then\_alerting}(t)$, $\mathsf{then\_connected}(t1, t2)$ as interface (output) variables. These variables are implicitly predicated over rounds (events) so they are just expressed as predicates over telephones. The implementation makes use of the fact that there can be at most one asking phone. Also, whenever any connection is requested, all onhook phones need to be alerted. So the system module just keeps track of the connections and the identity of the asking phone. Modeling the updates of these variables presents no challenges and is directly derivable from the specification. The value of $\mathsf{then\_connected}(t1, t2)$ follows simply from the connections for $t1$, $t2$ and $\mathsf{then\_alerting}(t)$ is true for any on-hook phone t whenever there is an asking phone.

To verify the implementation we need to prove (6) from Table 1 for our system (M, P) and specification (S). In reactive modules, every module variable must have a value in each round, regardless of input. So the consistency of the system

$$\exists\ sv\ sh.\ \mathsf{M} \wedge \mathsf{P}$$

is implicitly guaranteed by the programming platform. What remains is to prove that for all values of *ev* that do not falsify S, the following holds true:

$$(\forall sv\ sh.(\mathsf{M} \wedge \mathsf{P}) \Rightarrow \mathsf{S})$$

We write an module that generates a superset of the values of *ev* that do not falsify S and supply the generated *ev* as input to the system module. Then we prove the above property by composing the system module in parallel with the specification and checking that the invariant holds for all possible states.

We use the enumerative model-checker in the Mocha system for the proof. Since the model checker will only work for a finite state space, we need to fix the number of telephones. Most of the non-trivial conditions of the VTS become visible when we have more than four telephones. We verify the system for a village with up to 9 telephones.

We ran the enumerative model-checker in Mocha-1.0 on a 167MHz Sun UltraSPARC with 96MB memory, running SunOS 5.5.1. For a village with 6 telephones, the system has 233 reachable states and is verified in 80 seconds. At 9

telephones, it has 9497 reachable states and the verification takes 233 minutes. The model-checking breaks at 10 telephones for lack of memory. This suggests that Mocha is probably useful as a debugging aid, allowing non-trivial tests, but cannot handle the number of states involved in checking any but the smallest villages. Clearly it would be of interest to find "saturation" principles that allow us to conclude properties of villages of all sizes from those of a fixed size, or techniques for allowing the checker to be used in conjunction with infinitary proof techniques like induction.

8 Conclusions and Acknowledgements

We have shown how to carry out an "end-to-end" formal development of an illustrative software system. This process included modelling parts of the process that are not usually treated formally, such as the user-level requirements. By a systematic approach to refinement we have shown how these requirements can be reduced to a specification that a programmer can implement. Formal proofs were developed for each of the refinements involved except for a gap between our SML implementation and its (extremely similar) HOL specification, and a gap between our Mocha specification and a corresponding HOL specification. The benefit of closing the first gap is probably not worth the trouble in this case, but better integration between Mocha and HOL could yield interesting benefits.

We would like to express thanks to Rajeev Alur, Trevor Jim, and Insup Lee for their input to this work.

References

1. R. Alur and T.A. Henzinger. Reactive modules. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science*, pages 207–218, 1996.
2. 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.
3. Marco Defillers, David Griffioen, and Olaf Müller. Possibly infinite sequences in theorem provers: A comparative study. In *Lecture Notes in Computer Science 1275: Proceedings of the 10th International Conference, TPHOLs '97*. Springer, 1997.
4. Carl A. Gunter, Elsa L. Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. Available by request, 1998.
5. 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.
6. 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.
7. Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *Transactions on Software Engineering and Methodology*, 1998. To appear.