The Logic of Bugs

Gerard J. Holzmann
Bell Laboratories, MH-2C-522
600 Mountain Avenue
Murray Hill, NJ 07974, USA
+1-908-582-6335

gerard@research.bell-labs.com

ABSTRACT

Real-life bugs are successful because of their unfailing ability to adapt. In particular this applies to their ability to adapt to strategies that are meant to eradicate them as a species. Software bugs have some of these same traits. We will discuss these traits, and consider what we can do about them.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification D1.3 [**Programming Techniques**]: Concurrent Programming.

General Terms

Algorithms, Reliability, Theory, Verification.

Keywords

Distributed systems software, logic model checking, SPIN.

1. INTRODUCTION

The ability of bugs to adapt to their environment is well known. It typically takes only five to ten years for insects to become immune to specific pesticides, and in about the same amount of time viruses and bacteria can 'learn' how to defeat the drugs that are developed to kill them. These are, of course, not the traits of individual organisms, but they are the traits of a species. Using natural selection to pass information from one generation to the next, the species, as it were, can develop an ability to side step its attackers, and thrive.

There are some interesting parallels in this sense between software bugs and real-life bugs. Software bugs seem to have the same uncanny ability to defeat process improvement efforts that are meant to eradicate them.

After some fifty years of practice, few people today would say that we understand the problem of software quality well enough that we could outline a development process that could lead to zero-defect code. This is despite the fact that in the last fifty years the methods we have developed to prevent and intercept bugs have gotten significantly better. We have higher-level languages that can prevent many low-level mistakes; we have better compilers that can catch many coding mistakes through some forms of static analysis, and we have improved training for programmers. Still, those pesky bugs routinely manage to outsmart even the most experienced programmers.

Some of the ways in which bugs have learned to adapt to improved software development environments are:

- Bugs can adjust to the level of experience of the programmer. One common misconception is that experienced programmers make fewer mistakes than novice programmers. Experienced programmers and novice programmers make roughly the same number of mistakes when writing the same amount of code. The mistakes made by the experienced programmer, however, will be more subtle than those of the novice programmer. The more complex bugs that the experienced programmer can seed into the code are often harder to find than the simpler typos of less experienced colleagues.
- Bugs can invade our test environment. It has been said that the safest place for a fly to sit is on the fly-swatter (presumably on the handle…). If our test setup is itself buggy, we will not find the bugs we are looking for. Similarly, if the original *requirements* for the software are inaccurate, or the test objectives that are derived from the requirements, the remaining test process could be perfect, but it would still be ineffective.
- Bugs have developed a strategy to quietly replace any one of them that is caught immediately with one or more others. This is the well-known phenomenon of developers being most likely to insert new bugs into

Conference '00, Month 1-2, 2000, City, State.

their code while they are trying to fix it to remove old bugs. Bugs thereby counteract one of the primary means we have to eradicate them.

The effect that bugs have their easiest entry into code during a version update process is easily confirmed by looking at historical data for the distribution of the Spin model checker [5]. Figure 1 shows the frequency of the number of days that have passed between version updates of the Spin sources over a period of more than ten years. Several months can pass in between major updates of stable versions of the code (the largest interval to date being 180 days). If bugs creep in, despite best efforts to avoid this, this typically results in a new version release that follows a prior release within a few days. Since the same process repeats for every new release, over a sufficiently long time we would expect to see a relatively large number of version releases with short intervals, and a more random distribution of longer intervals between releases. Figure 1 illustrates this effect nicely. The shortest interval is 0 days, for one instance where two releases of Spin were issued on the same day (Versions 2.3.7 and 2.3.8 from May 18, 1995). Nearly half of all releases to date (65 of 141) followed the immediately preceding release within one week.

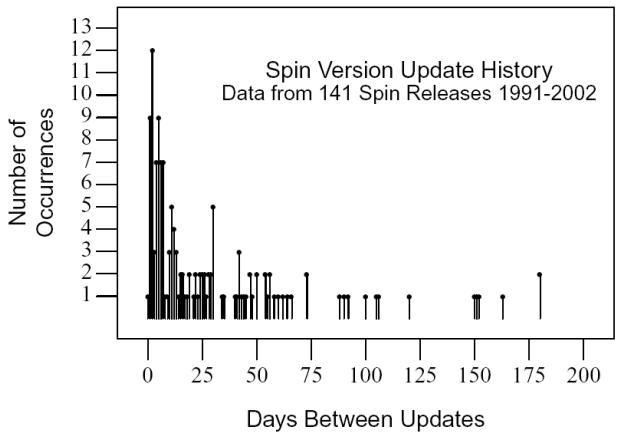

Figure 1 View of the Spin Release History

1.1 CAUSES

The main reason why all these problems exist is disappointingly simple: programming is a human activity and humans are prone to error. As Mark Paulk from Carnegie Mellon University's Software Engineering Institute dryly noted ``*A fundamental problem with software quality is that programmers make mistakes*'' [8]. The usual industry estimate is that a typical piece of code contains one to ten residual software defects per one thousand lines of code written (i.e., bugs that survive all testing). This means that if a product has one million lines of code, the end-user should expect roughly one thousand residual, or *latent*, faults. Not all of those faults will show up in practice of course, which may lead to a false sense of quality. A more popular product, though, will attract more users, who in a way act as volunteer testers. And, more testers will cause more latent faults to reveal themselves. This leads to the curious effect that the perceived quality of a product can *decrease* as the number of users of the software *increases* [7].

It is often suggested that software products are inherently less reliable than other types of products that are equally subject to human error. This supposition, however, does not seem to be supported by the available evidence. If, for instance, we consider programming to be a form of stylized writing, we can try to compare this activity with other forms of writing where correctness is important. An average weekday issue of The New York Times, for instance, contains 13,800 sentences [1]. Inevitably, some mistakes are made. In each issue the paper therefore also publishes a list of corrections to the previous issue. In a typical issue of the paper, there are about 10-15 of those corrections. This averages out to about one mistake for every one thousand sentences written. Not surprisingly, this is roughly the same residual defect ratio as is achieved by careful programmers. Journalists and programmers are subject to the same human fallibility.

Given the fact that programmers do make mistakes, and will likely continue to make mistakes no matter how hard they try to avoid doing so, we are stuck with the job of finding better ways to intercept those errors, hopefully in ways that software bugs will find harder and harder to survive.

1.2 OVERVIEW

The premise of this paper is that it is unlikely that we will ever come up with effective means that can prevent bugs from entering the software design process altogether. Given that, our best strategy is to devise means to detect the presence bugs as effectively as possible, so that they can then be removed. Bugs, though, can develop counter-strategies that we must be aware of. We will argue that the best means we have to defeat bugs is to use the computational power of computers against them. One of the techniques that allows us to do so is logic model checking. We will describe where bugs are most likely to find hiding places when these techniques are used, and how we can make it less likely that they will be successful in that attempt.

2. TRADITIONAL METHODS

We collectively have about half a century of experience with the job of producing and debugging software. One can expect that in that period of time the dust can pretty much settle on what the right way is of structuring software in such a way that the introduction of errors can be minimized, and in testing software in such a way that the

bugs that do slip into the code can be caught. What are the methods that are used in the top industries to solve these problems?

In the telecommunications industry a rather rigorous software development process was adopted in the seventies and eighties. The development starts with requirements capture and documentation by system engineers. That phase is followed by high-level and low-level design (i.e., programming), followed by various stages of testing, and finally customer acceptance. The final phase of the process is maintenance. In each phase errors can be introduced into the process and in each phase errors can be intercepted. The common wisdom is that the earlier a bug is intercepted, the less expensive it will be to fix it.

In the requirements phase, inconsistencies can be caught with basic requirements analysis techniques, in the high level design phases errors are caught by prototyping, simulation, and modeling, in the low level design phase peer review sessions and targeted unit testing techniques can be used. In the final testing phase this consists of integration and customer acceptance testing.

All these techniques do in fact combine to intercept the majority of all errors that are introduced. But these techniques do not bring the residual error rate down to zero. In all likelihood, not even the most diligent application of these traditional techniques *could* even bring the residual error rate down to zero. There are several reasons for this.

First consider sequentially executing deterministic software. To be of any interest at all, these programs must read their input date from some external source and use these data to compute a result that can then be returned to the user. In most cases it will be impossible to test the program across the entire range of possible inputs. So a sampling method has to be used. Again, in almost all cases of interest, the sampling method will be based on a heuristic and will be incomplete, leaving open the possibility of a missed error. Bugs prefer to hide in places where testers decide not to look, thus improving their chances of survival.

The situation is more interesting still for concurrent, and non-deterministically executing software. Almost all modern software falls into this category. This is an ideal place where bugs can currently hide safely, immune to most of the currently used testing strategies.

This time not just the input data can influence the computations but also the relative ordering of events that may occur in different parts of the system is important. The unpredictable ways in which multiple users can interact with the different parts of a physically distributed system become a factor, and similarly the unpredictable way in which schedulers run asynchronous threads of execution. Two issues complicate our ability to test or debug such systems: limited *observability* and limited *controllability*.

Limited observability means that the tester or the user cannot observe important details of an execution (such as the precise interleaving of multiple threads in an execution).

Limited controllability means that the tester cannot control those details either. Even if a precise interleaving that needs to be tested is known, it generally cannot be reproduced on a live system. The fundamental non-determinism that is exhibited by distributed systems makes it virtually impossible to test them thoroughly by traditional means. We face these problems in the testing of practically all systems of interest that are built today: operating systems, telephone systems, plant control systems, traffic control systems, web applications, etc.

3. AUTOMATA AND LOGIC

The traditional ways of testing software do not use the power of computers very well. Somewhat better in this sense are methods that try to automate specific parts of the design process. In this class we find methods for automatic test generation, regression testing, and code generation. These methods can bypass the error-prone human, and thus reduce the error insertion rate somewhat.

We can do still better though with methods that can directly take advantage of the steadily increasing power of computers. Many of the new methods in the latter class are based on the use of automata and logic in one form or another. This includes static analysis and symbolic interpretation methods, reachability analyzers and logic model checking tools. These tools try to overcome the limitations of traditional testing by constructing and analyzing closed models, or symbolic representations, of real-world artifacts. In most cases, these methods depend on an ability to either solve computationally hard problems, or to find reasonable approximations to their solution. That trait has sometimes been considered a weakness, but this weakness is disappearing as computers are becoming more powerful.

The phenomenon that bugs can invade any technique and make it less effective does of course still exist. We will consider below how a new strain of bugs is attempting to do so with the newer methods, and we will reflect on what we might be able to do about that.

3.1 COMPUTATIONAL COMPLEXITY

When reachability analysis techniques were first introduced for the verification of data communications protocols in the late 1970s and early 1980s, the most often cited problem was the problem of computational complexity. No paper was published in this period without a serious discussion of the so-called *state space explosion problem.*

Although many verification algorithms have a complexity that is only linear in the number of reachable system states, the spoiler is that the number of reachable system states can depend exponentially on the value of some critical problem parameters. Those parameters include the number of asynchronously executing processes, the number and length of message buffers, the effective range of data types, etc. Model checking tools would be quite useless in practice if the theoretical worst-case behavior would occur frequently enough. Fortunately, it occurs only rarely. The same holds for many other types of undeniably useful tools that solve computationally hard problems, yet that we have come to depend on for solving day-to-day problems. This class includes, for instance, compilers, graph layout tools, and network optimization tools. Luckily for us, in none of these cases do users worry much about theoretical worst-case behavior, simply because it does not occur often enough in practice.

The steady increase of computational power of average desktop machines combined with the steady improvement in algorithmic techniques that static analyzers and logic model checkers can use to reduce the complexity of typical verification problems have brought about notable changes. To counter an exponential effect one ideally needs to find another exponential effect that works in the opposite direction. The algorithmic improvements that have been developed in the last two decades (e.g., partial order reduction techniques, BDD based model checking techniques, and abstraction techniques) can offer such effects, e.g., [5]. The steady increase in the power of average computers adds an additional exponential counter-effect, acting quite independently of the first.

Since the nineteen-fifties the raw speed of computers has increased by about *ten orders of magnitude*. The average desktop PC that anyone can buy today is more powerful than the average supercomputer that very few large corporations could afford to buy just two decades ago. Differences of this magnitude have a dramatic impact on what can be done in software verification, and the continuing trend can very significantly influence what we can do tomorrow.

The often heard rebuttal to the observation about the continuing effect of Moore's law as first articulated in [9] is that the increase in complexity of software is outpacing the increase in the power of the machines that we can use to check that software. This notion, though, does not quite hold up. At the first conference on software engineering [10] (where the term *software crisis* originated) the size of one of the larger systems being developed at the time, IBM's system OS/360, was given as five million lines of assembly level code. The modern day equivalent would be approximately one million lines of C code. Today, 34 years later, a typical operating system contains less then one hundred million lines of code. This means that the size of large software applications has increased by less than *two* orders of magnitude in thirty years, while the compute power we can use to analyze that code has increased by *six* orders of magnitude. There is little doubt that an average compiler today can check ten million lines of code more thoroughly and more quickly than its ancestor could check a ten-thousand line program in 1970.

Computational complexity for model checkers such as Spin is close to becoming a non-issue. It seems unlikely that there is an effective adaptation strategy for bugs to evade the ever-increasing power of our analyzers. But, bugs can still evade our scrutiny by finding different places to hide, as we shall discuss next.

4. MODEL CAPTURE

Logic model checkers work with closed, finite models of real world artifacts. Traditionally, these models are constructed by hand, as prototypes of a design. The effort involved can be justified by pointing to similar efforts that are made in other engineering disciplines to obtain analytic results. The civil engineer may construct a small prototype of a new design to study its structural integrity, and the mathematician may construct a mathematical model that captures a simplified view of the real world that lends itself more easily to analysis. The software engineer will often also build a small prototype of a software design, to subject the prototype to simulation experiments and perhaps a usability study. So, the notion that one can first build and then analyze a design model in a special purpose modeling language is sound, well known, and useful.

When model checking is applied for software verification, there are two problems to be overcome. First, model building is a skill that must be learned. Model checkers may appear attractive at first because they seem to deliver push-button results, but the push-button results obtained by the first-time user are not nearly as impressive as those that can be achieved by more experienced users. Fortunately, novice users do tend to become experienced users over time, so this problem is only a temporary one.

A second, more basic, problem is that the method we sketched above only helps us to find bugs in high-level system designs, and we may be more ambitious than that. We may, for instance, also want to use these methods to find typical bugs in implementation level code: the purview of classic software testing techniques. This problem too may be overcome. We know that in some cases it is possible to extract verification models from implementation level code mechanically. Techniques to do so were developed for Java [3] and for C [6] and [2]. Not all technical details have been overcome in this area. Model extraction for a substantial piece of code requires the con-

struction of a test-harness, which is still a fairly non-trivial human effort. By using a combination of automated abstraction, model extraction, and logic model checking we may be able to increase the scope of applications sufficiently to make this approach to software verification more routine.

4.1 REQUIREMENTS CAPTURE

We now come to the third and most underestimated problem in applications of automated tools to software verification: the problem of accurately capturing the correctness requirements that have to be verified. Many users take this issue for granted: they would like their software application to be proven correct, without having to worry about what correctness actually means. Every verifier knows, though, that there is no such thing as absolute correctness. We may be able to prove that a software artifact has, or does not have, a specific set of explicitly stated properties, but whether all those properties taken together constitute correctness is an issue that a verification tool cannot settle: only a human user can try to do so (subject to error, of course). An application, for instance, cannot be presumed to be *correct* if a verification tool has merely proven that it is completely specified and cannot deadlock.

Verifiers, and logic model checkers in particular, excel in their ability to show that specific requirements are either satisfied or may be violated. It is up to the user to come up with a comprehensive list of requirements to be checked, and to make sure that the requirements that are checked make sense. That is a more difficult task than many realize, as we will try to illustrate below.

4.1.1 TEMPORAL LOGIC

The standard method to express the correctness requirements for a model checking application is to use the formalism of temporal logic, as first proposed for this purpose by Amir Pnueli in [11]. In Linear Temporal Logic (LTL for short), as used in Spin [5], a small number of special operators are introduced to allow one to reason about sequences of events that are claimed to be causally related.

LTL has the important benefit that it allows us to make precise, and often concise, statements about complex system properties. But, there is also a downside. To use LTL well requires a level of sophistication that many users never develop. The danger of improper use this time is not an unwarranted increase in the complexity of a verification run, it is that the user unknowingly obtains results that are invalid. A small example can illustrate this effect.

Suppose that we want to verify the following simple property of a telephone system:

''When the subscriber picks up the phone, dial-tone is always generated.''

The property is simple enough that it can be clearly stated in just a few words, and there will be no confusion whatsoever about what the informal statement actually means. This is different when we switch to temporal logic. Our first attempt can be to convert the requirement as directly as possible into an LTL formula, for instance as follows.

$$\Box\,(\mathit{offhook} \rightarrow \Diamond\ \mathit{dialtone})$$

In words, this formula says that it is *always* (the box operator $\Box$) the case that the occurrence of an *offhook* event *implies* (the logical implication symbol $\rightarrow$) that *eventually* (the diamond operator $\Diamond$) a *dialtone* event will occur. A verification run with this property, where we try to find any possible scenario that could violate this property, would most likely come up empty, and the (human) verifier could believe to have proven the property. But this would be quite incorrect.

Note that the property does not rule out the case that on a failure to hear a dial-tone a subscriber would return the phone *onhook* and try a new, this time more successful call. The *dialtone* would then be generated only if the *offhook* event is repeated, which is clearly not the intent.

We can try to counter this by slightly modifying the property, but we quickly notice that it is easier to reason about the type of error scenario that we want to prove to be *infeasible*, rather than the property itself. So let us switch to an LTL formula that captures the error behavior we are interested in. Taking into account our earlier mistake, the new formula could look as follows.

$$\Diamond\ (\ \mathit{offhook} \rightarrow (\ \neg\ \mathit{dialtone}\ \mathbf{U}\ \mathit{onhook}\ )).$$

In words, this formula states that *eventually* the occurrence of an offhook event could logically *imply* the *absence* of a *dialtone* event that may persist *until* (the until operator **U**) an *onhook* event occurs. Did we capture the right property this time? Unfortunately, the answer is no.

We have made a very common mistake here in confusing logical implication with causal implication. The arrow operator is *not* a temporal operator. The definition of logical implication is simply:

$$(p \rightarrow q) \equiv (\neg p \vee q)$$

The requirement expressed here is that in the current system state either p is *false* or q is *true*: there is no temporal relation here. This means that the last LTL formula we gave is already satisfied if at least once in an execution *offhook* does not occur (i.e., if offhook is *false*). For the same reason it is also satisfied if at least once in an execution an *onhook* event occurs (since the sub-formula (p U q) is immediately satisfied if q is *true*). This makes it possible to satisfy the formula quite independent of the *dialtone* event, which is again clearly not what we intended. Having realized the problem, it is again easy to correct it. We then

arrive at the following LTL formula, our third attempt to capture this simple correctness property:

$$\Diamond (\mathit{offhook} \wedge (\neg \mathit{dialtone}\ \mathbf{U}\ \mathit{onhook})).$$

This time we did capture the right behavior, but we also captured more than we intended. Note that the formula is already satisfied when *offhook* and *onhook* are simultaneously *true*. It would be acceptable to leave things this way, since clearly in the application we are interested in this cannot ever occur. But, for arguments sake we may also want to repair this flaw. We can do so by introducing the Next operator (represented by the symbol **X**).

$$\Diamond (\mathit{offhook} \wedge \mathbf{X} (\neg \mathit{dialtone}\ \mathbf{U}\ \mathit{onhook}))$$

We have now separated the state where we expect to find the *offhook* event from the subsequent states in which the *onhook* may take place, so all seems well. We now note that this last formula will erroneously be satisfied for an execution where a *dialtone* event could coincide with the *onhook* event. That is: an execution where a *dialtone* would be generated at the exact instant that the subscriber returns the phone *onhook* would incorrectly be flagged as an error. We can repair this flaw too by extending the formula as follows

$$\Diamond (\mathit{offhook} \wedge \mathbf{X} ((\neg \mathit{dialtone} \wedge \neg \mathit{onhook}) \setminus$$
$$\mathbf{U}\ (\neg \mathit{dialtone} \wedge \mathit{onhook})).$$

The ω-automaton [13] that is generated from this LTL formula by standard converters (such as the one that is built-in to the Spin model checker [5]) is illustrated in Figure 2.

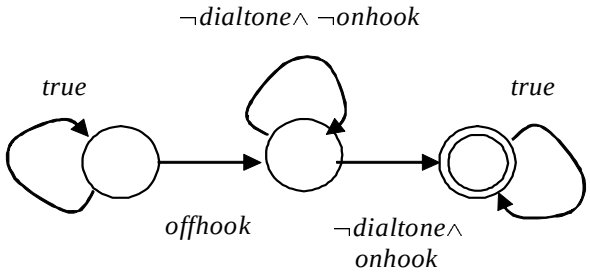

Figure 2 Omega Automaton for the Dialtone Property

It should be noted that even though coming up with an accurate formulation of this seemingly very simple error-condition is hard, it is still harder to come up with an accurate version of the *positive* system requirement that we started with (i.e., the logical negation of the last LTL property). The formal statement of that property requires the use of the temporal **V** operator (the dual of the **U** operator), which is defined as follows:

$$(p\ \mathbf{V}\ q) \equiv \neg (\neg p\ \mathbf{U}\ \neg q).$$

It can be quite difficult to develop a good intuition for these types of formulae.

4.2 VISUAL FORMALISMS

The automaton structure shown in Figure 2 looks more intuitive than its equivalent in LTL, but this too can be misleading. It can be difficult as well to specify complex requirements accurate when directly coding an ω-automaton structure. In the first few distributed versions of the Spin model checker this was the primary means for specifying temporal properties. The addition of support for LTL syntax in Spin version 2.7 was seen by many users as a notable improvement. So if automata structures are not an adequate formalism, and temporal logic is not quite adequate, what alternatives remain?

One proposal has been to prepare a library of predefined LTL formulae from which a user can choose, based on short template descriptions of each basic type of property [4]. This approach is often effective, but it cannot avoid cases where incomplete intuition leads to the selection of either an incorrect or an incomplete pattern. For the *dialtone* property, for instance, the first pattern that would seem to match would be an ***Absence*** property, where we look for the absence of *dialtone* in between an *offhook* and an *onhook* event. The pattern for this in the database formalizes absence of *P* in between *Q* and *R*:

$$\Box(Q \wedge \neg R \wedge \Diamond R) \rightarrow (\neg P\ \mathbf{U}\ R))$$

replacing the symbols *P* with *dialtone*, *Q* with *offhook* and *R* with *onhook* then gives:

$$\Box\ (\mathit{offhook} \wedge \neg\ \mathit{onhook} \wedge \Diamond\ \mathit{onhook}) \rightarrow\ \setminus$$
$$(\neg\ \mathit{dialtone}\ \mathbf{U}\ \mathit{onhook})).$$

This formula looks intimidating enough that most users would not question its validity. But it is still incorrect. Note, for instance, that the formula is satisfied for any state where *offhook* is *false* or where *onhook* is *true*, quite independent of any occurrence of a *dialtone* event. Because of the incorrect usage of the logical implication symbol, the formula has still more problems. Note that it is also satisfied for executions where *onhook* remains invariantly *false*.

We have pursued an alternative approach, based on the development of an intuitive graphical editor for the specification of basic temporal properties [12]. The types of requirements that can be specified in this way are more limited than the full use of LTL allows, but this appears not to be a significant restriction. (There may actually be some benefit in avoiding the full complexity of LTL.) The correct version of the *dialtone* property, for instance, is specified with the TimeLine editor as illustrated in Figure 3.

The timeline shows two events of interest, *offhook* and *onhook*, on the timeline. The executions of interest are identified by the initial event, which is marked with 'e' for event. An arbitrary number of constraints to the execution can be given below the timeline, over specific bounded intervals of the execution. Here we have used a single constraint to state that we are only interested in executions where no *dialtone* is generated after the initial *offhook*. An error condition is detected if under these circumstances an *onhook* event can occur: which is indicated by placing the *onhook* event on the timeline as a failure event, marked with the symbol 'f' and a cross.

Of course, also the intuition that is supported by timeline specifications cannot prevent occasional mistakes. The three events of interest in the *dialtone* property, for instance, can be assigned in several different ways to constraints underneath the timeline, or to optional, required, or failure events within the timeline. Unlike the various forms of LTL formulae, though, each such variant of a timeline is visually no more complex than any other. The hope is that this visual simplicity allows the user to concentrate on the meaning of a property, rather than on the subtleties of its formalization. To support that intuition, the user can inspect the precise structure of the ω-automaton that corresponds to the timeline as specified. The timeline shown in Figure 3 precisely corresponds to the ω-automaton from Figure 2.

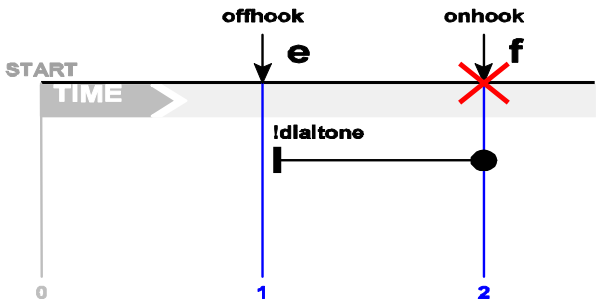

Figure 3 Timeline Specification of the Dialtone Property

5. CONCLUSIONS

Human work is subject to error. Humanly produced software, therefore, unavoidably contains mistakes. Many of these mistakes can effectively be intercepted with the help of traditionally used tools, ranging from compilers to static analyzers and runtime checkers. We know of only one effective method that can be used to intercept concurrency related errors though: logic model checking. We claim that any checking methodology that can directly exploit the steady improvements in computational power will ultimately gain a decisive advantage over methods that do not have this capability.

But also in applications of logic model checking errors can creep in. For instance, when the formal models are manually constructed, bugs can enter the models and hide requirement violations from view. We can mitigate the effect of this by using techniques for automated model extraction, e.g., as in [3], [6], and [2]. There are still some manual steps here, for instance in the definition of the test-harness or of the type of abstraction that is used, but the opportunity for undetected error is further reduced.

With a software verification process based on model extraction and logic model checking techniques, the one final place where bugs can comfortably hide is in the property definitions. The formulation of the requirements for a software artifact is a fundamentally human task, and as such unavoidably subject to human error. The commonly used formalism of Linear Temporal Logic can challenge the (human) verifier's intuition, which can seriously jeopardize the validity of verification efforts. As a countermeasure, we have discussed the use of a simple visual formalism that can support our intuition more effectively. The lesson learned from these tools is that it can sometimes be wise to trade generality for confidence.

6. ACKNOWLEDGEMENTS

Margaret Smith developed the Timeline Editor discussed in this paper, jointly with Kousha Etessami and the author.

7. REFERENCES

[1] L. Amster, D.L. McClain (Eds.), Kill Duck Before Serving, Red Faces at The New York Times, Publ. St. Martin's Griffin, New York, 2002, 172 pgs.

[2] T. Ball, and S.K. Rajamani, Boolean programs: a model an process for software analysis. MSR Technical Report 2000-14, February 2000, 29 pgs.

[3] J. Corbett, M. Dwyer, et. al, Bandera: Extracting finite-state models from Java source code., Proc. ICSE 2000, Limerick, Ireland.

[4] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In Proc. 21st Int. Conf. on Software Eng., ICSE99, Los Angeles, May 1999. See also: http://www.cis.ksu.edu/santos/spec-patterns/ltl.html

[5] G.J. Holzmann, The model checker Spin. IEEE Trans. On Software Engineering. May 1997, Vol. 23, No. 5, pp. 279-295.

[6] G.J. Holzmann, and M.H. Smith, Software model checking: extracting verification models from source code. Formal Methods for Protocol Engineering and Distributed Systems, Oct. 1999, Kluwer Academic Publ., pp. 481-497.

[7] G.J. Holzmann, Economics of Software Verification, Proc. ACM Workshop on Program Analysis for Soft-

ware Tools and Engineering, Snowbird, PASTE2001, Utah, USA, June 2001.

[8] Information Week, Issue on Software Quality, Jan. 21, 2002.

[9] G.E. Moore, Cramming more components onto integrated circuits. Electronics, April 19, 1965, pp. 114-117.

[10] P. Naur, and B. Randell (Eds.) Report on Working Conference on Software Engineering, 1968, Garmish, Germany. Publ. 1969, NATO, Brussels.

[11] A. Pnueli, The temporal logic of programs. Proc. 18th IEEE Symposium on Foundations of Computer Science, 1977, Providence, R.I., pp. 46-57.

[12] M.H. Smith, G.J. Holzmann, and K. Etessami, Events and constraints, a graphical editor for capturing logic properties of programs. Proc. 5th Int. Symp. on Requirements Engineering, Toronto, Canada, August 2001, pp. 14-22.

[13] W. Thomas, Automata on infinite words. Handbook on Theoretical Computer Science, Volume B, Elsevier Science, 1990, pp. 135-165.