Formalizing Requirements is $\Diamond\Box$ Hard

Gerard J. Holzmann

Nimble Research, Monrovia, CA 91016, USA
gh@nimbleresearch.com

Abstract. The use of formal methods in software engineering requires that the user adopt of a suitable level of precision in the description of both design artifacts and the properties that should hold for those artifacts. The level of precision must be sufficiently high that the logical consistency of the design and the logic properties can be verified mechanically.
The source code of any well-defined program is itself a formal object, although it typically contains more detail than desirable for effective analysis. But, practitioners often have no problem producing or recognizing an abstracted version of the key features of a design, expressed in the modeling language of a verification tool.
The real problem preventing a broader acceptance of formal methods is that there are no intuitive formalisms that practitioners can use to express logic requirements at the level of precision that is required for formal verification. That problem is the focus of this paper.

Keywords: Software verification · Logic model checking · Temporal Logic · Rule based specification

1 Introduction

Some claim that all defects in software products find their root cause in misstated requirements, e.g., [1]. Industrial software development processes typically start with a requirements elicitation process, with requirements captured in English prose documents. The requirements are meant to be understood with some ease by software developers, who can fill in gaps and resolve cases of ambiguity by using common sense and experience. But, as we know, this process can lead to problems.

A formal methods based process requires us to make things sufficiently precise that requirements and a suitably abstract representation of the code can be checked for their logicall consistency by purely mechanical means: by an algorithm. We will side-step the question here what the best way is to produce this magical "suitably abstract representation" of the code, and for convenience consider this a solved problem for now. What we would like to focus on is the difficulty of turning ambiguously and incompletely stated requirements into precise logic statements that can be verified mechanically.

2 The Charm of Informality

”When you pick up the phone, you get a dialtone.” Unfortunately, the relevance of this informal requirement is familiar only to a rapidly shrinking audience, but for those of us with longer memories, the meaning of the statement will be clear. The English text, though, is not precise enough for use in any formal verification tool.

Pnueli was the first to point out, in 1977 [2], that statements like these can be stated more precisely in temporal logic. A first attempt to do so can then look as follows:

$$\text{offhook} \rightarrow \Diamond \text{ dialtone}$$

where the right arrow $\rightarrow$ stands for logical implication, and the diamond operator $\Diamond$ is pronounced ”eventually.” But that formalization needs quite a bit more work before it could be considered correct.

The statement has precisely defined semantics that can roughly be summarized as saying: ”either the phone is not offhook now (i.e., in the *initial* state of the system we are considering) or it must be true that now or at some point in the future a dialtone will be present.” And, that’s not what we meant.

A first improvement is to make it clear that the property should not just hold when coincidentally the phone happens to be offhook in the initial state, but it should hold at any time. That leads to this version:

$$\Box \text{ (offhook} \rightarrow \Diamond \text{ dialtone)}$$

This is better, but still not quite right. Note that the property will be satisfied even if the dialtone is already present in the very state that the phone goes offhook. That takes away the cause and effect that we would like to verify, so we have to separate the effect from the cause. That leads to this version:

$$\Box \text{ (offhook} \rightarrow \text{X} \Diamond \text{ dialtone)}$$

where we used the X symbol to represent the ”next” operator, as we do in the Spin model checker [4].

Are we done now? No, not really. For a more complete statement we may also have to state that the dialtone should *not* be generated unless there was an offhook event first, and it also need not be generated if the offhook event is followed by an onhook event before the dialtone appears. Then we haven’t said anything about the requirement that the dialtone should be generated within the first few milliseconds after the offhook event, and it should disappear after the first digit is dialed.

Once we are done with a precise statement of this still relatively simple requirement, any user uninitiated in the use of temporal logic is unlikely to recognize the statement, nor would that user be likely to produce it.

2.1 Visualizing Requirements

One way to address these problems is to build tools that can visualize requirements in a more intuitive way, leaving it to a tool to synthesize the correct formalization of the requirement with all subtleties properly addressed.

One such tool that we experimented with long ago is the timeline editor [3]. The editor allowed users to place events of interest on a timeline, and to annotate that timeline with various types of constraints that can span overlapping periods of time. The tool could convert the specifications into Büchi automata, which could then be used directly in a logic model checking tool such as Spin [4]. A specification of the dialtone property is illustrated in Figure 1.

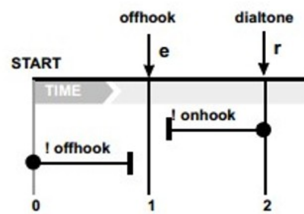

Fig. 1. Visual formalism for specifying temporal logic properties from [3].

The use of a visual formalism as shown here does still have limitations. One main drawback is, for instance, that the visual formalism is not expressive enough to specify everything that can be specified in temporal logic.

But then again, also temporal logic is not expressive enough to formalize all types of requirements that may be encountered in practice. The next section will give an example.

3 Graphs and Sets

The following example property is inspired by [6].

Consider a graph where each node represents a lock. There is an edge in the graph from node a to node b if at any time during an execution a thread holds lock a while it acquires lock b. If this graph contains a cycle, there is a potential for a deadlock scenario. Two events are of interest: *acquire(t,a)*, which represents the acquisition of lock a by thread t, and *release(t,a)*, which represents the release of lock a by thread t.

The property of interest is that in any given run of the system, the lock graph as defined above is acyclic. How can we express this formally?

The first problem we have is that basic temporal logic does not have variables, and it does not allow us to *bind* variables to symbols. If, for instance, we wanted to state that every time lock a is acquired it is also released, we have to be able

to correlate the a from the *release* event with the a from the corresponding *acquire* event. We may not know a priori which lock names might be used.

There have been a few attempts to explore extensions of temporal logic in model checkers, but the restrictions are usually non-trivial, cf. [5]. But even if we allow for variable binding, and add quantification, it is still very hard to talk about the existence of a cycle in a graph of unknown size.

It is possible to state a highly simplified version of the problem, if we restrict the problem to two threads and two locks only, by saying:

$\forall t_1, t_2, l_1, l_2 \cdot$
$\Box$ ((aquire(t_1, l_1) $\wedge$ ($\neg$ release(t_1, l_1) U aquire(t_1, l_2))) $\rightarrow$
$\Box\neg$ (aquire(t_2, l_2) $\wedge$ ($\neg$ release(t_2, l_2) U aquire(t_2, l_1))))

The formalization is very limited, but already relies on powerful features that are not present in most systems.

A very similar specification problem would be to formalize the type of check that is performed by the Eraser algorithm [7] for detecting data races in a multi-threaded program. In this case we want to maintain a lockset for every shared data object, and on every write access take the intersection of the lockset that is currently held by the writer with previous locksets. If the stored set every becomes empty we know there is a problem.

Here too the check asks us to track and update data, this time not in the form of a graph but as a set of sets (one for each shared data object). Temporal logic is clearly not equipped to handle this, but the question is: what type of formal specification could do so elegantly?

One method, described in the context of runtime verification tools, in an early draft of [6], is to use rule-based checks. We then minimally need the basic operations to insert or remove items from sets, and to check for set membership. Basic events like *aquire* and *release* then trigger set operations. Next we can define a set of rules to deduce facts of interest from the stored sets, e.g. using the Rete algorithm [8].

Writing properties in this way requires a change in perspective. Rather than a logic-based approach we now have a more operational view. Basically, we are writing a program to perform the checks we are interested in, though at a relatively high level of abstraction.

The deadlock problem we started this section with can be formalized in the following five rules (with minor edits, from the draft of [6]):

1. acquire(t, l) $\rightarrow$ insert(Locked(t, l))
2. release(t, l) $\rightarrow$ remove(Locked(t, l))
3. Locked(t, l_1) $\wedge$ acquire(t, l_2) $\rightarrow$ insert(Edge(l_1, l_2))
4. Edge(l_1, l_2) $\wedge$ Edge(l_2, l_3) $\wedge$ $\neg$Edge(l_1, l_3) $\rightarrow$ insert(Edge(l_1, l_3))
5. Edge(l, l) $\rightarrow$ error

This is a remarkably succinct specification, though, like specifications written in temporal logic, it may still require a substantial effort to create.

The last two rules perform what is basically term rewriting on the generated sets. Clearly, the computation of transitive closure in rule 4 and the indirect check

for cycles in rule 5 could be computationally demanding. To use this approach in a model checker, e.g., integrated with a depth-first search algorithm, we'd need to make arrangements for backtracking of changes to the constructed sets.

3.1 Rule-Based Model Checking

In principle, one could write an entire logic model checking procedure as a rule based system. After all, in standard explicit state model checking we merely construct a graph and check for the existence of specific types of cycles in that graph. There have been several attempts already to implement model checkers as rewrite systems, starting more than two decades ago, e.g. [9].

Yet, we are left with the question if rule based specification formalisms, or rewrite systems in general, are any easier to use than logic based formalisms?

4 Conclusion

There may be two factors that can explain the continuing reticence of software developers and software development organizations to adopt formal methods as a routine part of software design, even for safety critical applications as used, for instance, in cars and in medical devices. One factor is certainly the unpredictability of the demands on both human and computer time that a formal verification approach can impose. Cloud-based solutions may be able to change at least some of these trade-offs [10].

A more serious factor is that writing formal specifications is not just perceived to be hard, it actually is hard. Even expert users can be misled by the exact meaning of complex formulae stated in temporal logic. It is perhaps also telling that semi-formal methods based tools, such as static source code analyzers, that make use of predefined specifications have indeed found broad acceptance, and are routinely used in industrial software development. Routine use of these tool could be boosted still further with the advent of interactive code exploration algorithms for large code bases [11].

Logic model checking and formal program analysis techniques though seem stuck at the hurdle of making formal specification human friendly. What can we do to make formal requirements specification $\Box\Diamond$ Easy?

References

1. R.R. Lutz, "Analyzing software requirements errors in safety-critical embedded systems," Proc. IEEE Int. Symp. on Requirements Engineering, Jan. 1993, San Diego, CA, pp. 126-133.
2. A. Pnueli, "The temporal logic of programs," Proc. 18th FOCS, Providence, RI, Nov. 1977, pp. 46-57.
3. M.H. Smith G.J. Holzmann and K. Etessami, "Events and Constraints a graphical editor for capturing logic properties of programs," Proc. 5th Int. Symposium on Requirements Engineering, pp. 14-22, Toronto Canada, Aug. 2001.

4. G.J. Holzmann, "The Spin Model Checker – Primer and Reference Manual," Addison-Wesley, Mass. (2004).
5. J. Bohn, W. Damm, O. Grumberg, H. Hungar, K. Laster, "First-order-CTL Model Checking," Proc. Int. Conf. on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 1998, LNCS Vol. 1530, pp. 283-294, Springer, Berlin, Germany.
6. K. Havelund, G. Reger, D. Thomas, E. Zălinescu, "Monitoring Events that Carry Data," In: Lectures on Runtime Verification - Introductory and Advanced Topics, LNCS 10457, Springer (2018).
7. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, "Eraser: a Dynamic Data Race Detector for Multithreaded Programs," ACM Trans. on Computer Systems, 15:4, Nov. 1997, pp. 391-411.
8. Rete algorithm, `https://en.wikipedia.org/wiki/Rete_algorithm`
9. M. Clavel, S. Eker, P. Lincoln, and J. Meseguer, "Principles of Maude," In 1st Int. Workshop on Rewriting Logic and its Applications. Electronic Notes in Theoretical Computer Science, Vol. 4. 1996.
10. G.J. Holzmann, "Cloud-based Verification of Concurrent Software," Proc. VMCAI 2016, Springer-Verlag, LNCS, Jan. 2016.
11. G.J. Holzmann: "Cobra: a light-weight tool for static and dynamic program analysis," Innovations in Systems and Software Engineering, a NASA Journal, Vol. 13, No. 1, pp 35-49, March 2017. `http://spinroot.com/cobra`.