Unit Lemmas for Detecting Requirement and Specification Flaws

Ashlie B. Hocking
Dependable Computing
Charlottesville, VA, USA
ben.hocking@dependablecomputing.com

Jonathan C. Rowanhill
Dependable Computing
Charlottesville, VA, USA
jonathan.rowanhill@dependablecomputing.com

Ben L. Di Vito
Dependable Computing
Williamsburg, VA, USA
ben.divito@dependablecomputing.com

***Abstract*—Unit lemmas and a checklist of questions used to generate them can identify flaws in formal requirements and specifications early in the design process, reducing the overall cost and increasing confidence in the final product. We demonstrate how we can apply unit lemmas and the checklist to a tri-valued logic system.**

Keywords—formal verification, formal requirements, formal specification

I. Overview

One goal of unit tests in software development is to verify that software satisfies formal specifications [1]. This goal can be met in some programming languages such as SPARK Ada by formal proof of software correctness with regards to the formal specifications [2]. However, these approaches rely on the formal specifications themselves being correct.

Formal specifications can be verified against higher-level formal requirements [3]. This approach enhances confidence but relies on the formal requirements being both correct and sufficient for detecting flaws in the formal specifications.

We propose using *unit lemmas* to supplement verification of formal requirements and specifications. Unit lemmas are small lemmas designed to verify particular desired qualities about a requirement or specification. In addition to their primary purpose in verifying the desired qualities, unit lemmas can also serve as helper lemmas when proving higher-level properties. While the approach detailed here focuses on functional requirements and specifications, the approach should also be applicable to many nonfunctional cases.

The proposed benefit of unit lemmas is similar to the proposed benefit of unit tests. Unit lemmas should be able to reduce costs and provide higher levels of assurance by detecting flaws earlier in the development process.

II. Common Categories of Unit Lemmas

To verify a formal requirement or specification does not have logical flaws, we have created a checklist of thirteen questions to ask about functional specifications:

Reflexivity/Irreflexivity: For a *predicate* p, should p(A, A) return true (or false) for all A?

Symmetry/Asymmetry: For a *predicate* p, should p(A, B, …) return the same result as p(B, A, …)? Should p(A, B, …) return the opposite result as p(B, A, …)?

Transitivity: For a *predicate* p, does p(A, B, …) and p(B, C, …) imply p(A, C, …) for all A, B, C?

Commutativity: For a function f, should f(A, B, …) return the same result as f(B, A, …)?

Degeneracy: For a function over two or more objects of the same type, is there a consistent result to expect (e.g., 0) when two of the objects are identical?

Identity: For a function f, is there an I such that f(A, I) = f(I, A) = A for all A?

Absorption: For a function f, is there a 0 (e.g., false) such that f(A, 0) = f(0, A) = 0 for all A?

Associativity: For a function f, should f(A, f(B, C, …), …) = f(f(A, B, …), C, …) for all A, B, C?

Inverse: For a function f, is there an inverse function g such that for all inputs A, f(g(A))=g(f(A))=A?

Range: Should the result of a specified function fall into a particular range (e.g., only positive values), where this range is not encoded into the return type?

Composition: Should a predicate over two objects always be true (or always false) when one argument (e.g., point) is a component of the other (line)?

Alternate Formulation: Is there an alternate formulation of the specification component that is equally valid?

Logical Contradiction: Can the specification be used to prove *false* is true? If a specification can be used to prove *false*, then we know a flaw exists.

These questions are motivated by considering functional specifications, so checklists of other questions should be created for non-functional specifications.

III. Example

An example to demonstrate the application of unit lemmas is tri-valued logic, for which we will be using the PVS language to specify. There are many types of tri-valued logic, but the one of interest here is when you have a type that is one of "known true", "known false", or "unknown". This logic can be useful when dealing with measurement uncertainty, so for example, when asking if A < B, if A is 2.1 ± 0.05 and B is 2.2 ± 0.6, the answer would be "unknown". In this particular logic, we are not considering quantum mechanical situations, so the assumption is that A is either less than B or not, but we lack sufficient knowledge to determine the answer to A < B.

In PVS, we might define a type tril as:

```
tril: TYPE = {TRIKNOWNTRUE, TRIKNOWNFALSE,
        TRIUNKNOWN};
t?(t: tril): boolean = (t = TRIKNOWNTRUE);
f?(t: tril): boolean = (t = TRIKNOWNFALSE);
u?(t: tril): boolean = (t = TRIUNKNOWN);
```

Before defining specifications returning this type, we consider operations *on* this type. For example, we consider two types of equality: the built-in equality (which returns true if the two tril have the same literal value), and a tri-valued equality:

```
eqt(a, b: tril): tril =
  IF t?(a) AND t?(b) THEN TRIKNOWNTRUE
  ELSIF f?(a) AND f?(b) THEN TRIKNOWNTRUE
  ELSIF u?(a) OR u?(b) THEN TRIUNKNOWN
  ELSE TRIKNOWNFALSE
  ENDIF;
```

Here two tril are known to be the same if they are both known to be true or known to be false. If either one is unknown, then we do not know if they are truly the same.

As this operator is being considered as an "equality", we need to determine whether the operator is a logical equivalence. I.e., does the operator have the property of reflexivity, symmetry, and transitivity? First, we have to clarify the question somewhat, as this operator returns a tril and not a boolean. In considering the desired system, we determine we want the following lemma to be true:

```
tril_equals_reflexive: LEMMA
  FORALL(a: tril): t?(eqt(a,a));
```

In other words, we want a *known true* result that a tril equals itself. However, this lemma cannot be proven because when a is *unknown*, eqt is *not* reflexive. Note that the built-in equality is reflexive but also returns true for two separate values of *unknown* truth, which is not what we desire.

Perhaps we can convince ourselves we do not need that equality, and instead consider tril *not*, *or*, and *and* operators. The *not* definition adds to the boolean one that *not unknown* is *unknown*. The *or* definition is such that if either value is *known true*, the result is *known true*, if both values are *known false*, the result is *known false*, and otherwise the result is *unknown*. Similarly, *and* is defined such that if either value is *known false*, the result is *known false*, if both values are *known true*, the result is *known true*, else the result is *unknown*. These operators have the basic properties we want: *not* is its own inverse, *or* and *and* are symmetric and associative, have the expected degeneracy result ($or(a, a) = and(a, a) = a$), and have the expected relationships with *known true* and *known false*, e.g., $or(a, known\ false) = a$. The boolean *or* and *and* operators by themselves do not have an inverse, but they are related to themselves and *not* by De Morgan's laws. The tril versions also obey a tril version of De Morgan's laws:

```
DeMorgan1: LEMMA FORALL(a, b: tril):
   nott(ort(a, b)) = andt(nott(a), nott(b));
DeMorgan2: LEMMA FORALL(a, b: tril):
   nott(andt(a, b)) = ort(nott(a), nott(b));
```

If we are able to show that the law of the excluded middle is also known to hold, then that might be sufficient to mitigate concerns about not having a good equality operator. Unfortunately, if a is unknown, then $a \ or \ \neg a$ is not known to be true by the logic we have defined so far, as $\neg a$ is also unknown, and *or* of two unknowns is unknown. Additionally, if we define an axiom of the excluded middle, then we are able to prove that *false* is true.

Consider also a definition of *implication* for tril:

```
impliest(a, b: tril): tril =
  IF f?(a) OR t?(b) THEN TRIKNOWNTRUE
  ELSIF t?(a) AND f?(b) THEN TRIKNOWNFALSE
  ELSE TRIUNKNOWN
  ENDIF;
```

This definition matches boolean implication, except the function returns *unknown* for cases where if a is *known true* and b is *unknown*, if a is *unknown*, and b is *known false*, or if both are *unknown*. Unfortunately, this definition fails the first of our unit lemma questions: *reflexivity*. For our logic to be useful, *a should always imply a*.

We also considered other tri-valued logics that might have the properties we desired, including one based off quantum mechanics, and also explored building the uncertainty into the inequality operations directly. Consider the following generic definition of a non-deterministic operation:

```
NDOP: TYPE = [a: real, da: posreal,
        b: real, db: posreal -> boolean];
```

This definition takes two arguments to be compared, a and b, and their uncertainty, da and db, and then returns a boolean result. We can use this generic definition to define a less-than operator:

```
NDLT: {op: NDOP |
  FORALL(a: real, da: posreal,
      b: real, db: posreal):
    ((a + da) < (b - db)) IMPLIES
     op(a, da, b, db) = TRUE AND
    ((a - da) >= (b + db)) IMPLIES
     op(a, da, b, db) = FALSE};
```

This non-deterministic less-than will return true if a is definitely less than b and will return false if a is definitely greater than or equal to b but is otherwise unspecified. However, when we apply the unit lemma approach, we find that we are *not* able to prove the irreflexivity lemma:

```
NDLT_irreflexive: LEMMA
  FORALL(a: real, da: posreal):
    NOT NDLT(a, da, a, da);
```

Specifically, given the deliberate under-specificity of NDLT, we can prove this lemma neither true or false. Additionally, we find that we are also unable to prove transitivity for NDLT:

```
NDLT_transitive: LEMMA
 FORALL(a: real, da: posreal, b: real,
      db: posreal, c: real, dc: posreal):
   (NDLT(a, da, b, db) AND NDLT(b, db, c, dc))
   IMPLIES NDLT(a, da, c, dc);
```

As with irreflexivity, we can neither prove nor disprove this lemma. Importantly, these unit lemmas reveal flaws in our specification that we might not otherwise have noticed.

IV. Conclusion

Unit lemmas have the potential to provide to formal requirements and specifications benefits similar to unit tests for software implementation. Unit lemmas allow practitioners to detect flaws in formal requirements and formal specifications earlier than might otherwise be detected, including possibly detecting fundamental design flaws that might go undetected until after a product is released. Applying unit lemmas to formal methods should increase confidence in the correctness of a product while reducing the overall cost.

Acknowledgment

This work was funded by USAF AFRL/RQQA contract FA8650-17-F-2220. Approved for Public Release: Distribution Unlimited (Case Number: 88ABW-2020-2414).

References

[1] Y. Cheon and G. T. Leavens, "A simple and practical approach to unit testing: The JML and JUnit way," European Conference on Object-Oriented Programming, 2002.

[2] N. Kosmatov, C. Marché, Y. Moy, and J. Signoles, "Static versus dynamic verification in Why3, Frama-C and SPARK 2014," Int'l. Symposium on Leveraging Applications of Formal Methods, 2016.

[3] C. Heitmeyer, J. Kirby, and B. Labaw, "Tools for formal specification, verification, and validation of requirements," COMPASS'97: 12th Annual Conference on Computer Assurance , 1997