Providing Early Warnings of Specification Problems

Dustin Hoffman, Aditi Tagore, Diego Zaccai, and Bruce W. Weide

Department of Computer Science and Engineering
The Ohio State University
Columbus, Ohio 43210, USA
{hoffman.373,tagore.2,zaccai.1,weide.1}@osu.edu

Abstract. A formal software verification system relies upon a software engineer writing mathematically precise specifications of intended behavior. Humans often introduce defects into such specifications. Techniques and tools capable of warning about common defects can help them develop correct specifications by finding subtle issues that would permit unintended behavior. New specification-checking techniques and a tool that implements them, *SpecChec*, are described.

1 Introduction

A formal verification system normally tries to prove that implementations satisfy formal specifications, but it cannot show that those specifications properly capture informal requirements. Therefore, specifications that erroneously encode requirements, combined with verified implementations of those specifications, could lead to false confidence in a system. Deciding whether specifications match requirements is a problem that generally has been regarded as beyond the scope of formal methods because, essentially by definition, formal specifications are developed from *informal* requirements. However, some specification defects can be caught by performing internal consistency checks on specifications alone (see [1]). This paper expands upon previous work in automatic detection of specification errors by describing a new specification-checking technique along with the implementation of a tool, *SpecChec*, capable of detecting certain kinds of inadmissible specifications as described here and in previous work.

Our experience with writing formal specifications suggests that simply identifying which parameters might be modified by an operation is generally an easy part of the process of formalizing requirements. Writing a sensible post-condition is normally more prone to both logical errors and typos. So, if a parameter has been identified as one that might be (i.e., is intended to be) modified by the operation—but the post-condition as written admits a correct implementation that *never* changes it—the postcondition is probably wrong. We submit that such a specification should receive what we call a **trivial-update** warning.

For example, consider fire-control software for a torpedo. The launch system needs to increase the pressure in the tube to equalize with sea pressure.

J.M. Badger and K.Y. Rozier (Eds.): NFM 2014, LNCS 8430, pp. 92–97, 2014.

The design of this system might involve an operation `IncreaseP`. If its specification inadvertently allows a correct implementation to do *nothing* to the pressure (even though the pressure parameter is marked as modifiable), the torpedo might not be able to launch. Thus a verified system fails.

Our proposed technique identifies trivial-update defects when a specification is being created. Typically, errors are detected by a verification system (in our case, the RESOLVE tools [2]) when a verification condition (VC) cannot be discharged by a theorem-prover (in our case, Z3 [3] and/or SplitDecision [4]) and subsequently the VC is traced back to its origin in implementation code. The specification-checking approach used here also involves generating VCs and feeding them to a theorem-prover—but much earlier in the design process. Since the cost of detecting and fixing errors increases as software development reaches the later stages of its life-cycle, eliminating errors early is widely regarded as a best practice in software engineering [5].

We assume familiarity with formal design-by-contract specifications using pre- and post-conditions. However, except as noted here, no prior knowledge of the RESOLVE language [6] or tools [2] or any specific theorem-prover is necessary. The ideas apply to specification and verification more generally.

The primary contributions of this work are in codifying and automating certain checks on the quality of specification engineering:

- A technique to detect trivial-update defects: specification errors that admit unintended behavior, i.e., failure of the specification to capture requirements.
- A tool to warn the developer of this and other specification defects.

2 Specification Modes

Most formal specification languages have syntax to mark each formal parameter whose value might be changed by an operation, such as the `modifies` clause in Dafny [7], JML [8], and Spec# [9]; by default, a parameter's value is unchanged. RESOLVE uses a slightly different approach. Every formal parameter to an operation has a **specification mode** ("mode" for short) that concisely declares something about how its value is affected by the operation, and about the relevance of the incoming and/or outgoing value of that parameter to the operation's overall effect. There are multiple modes in RESOLVE, four of which are of particular interest for this paper: `restores`, `updates`, `replaces`, and `clears`. The latter three can be viewed as specializations of the all-encompassing `modifies` of other languages. The mode `restores` indicates that the outgoing value is the same as the incoming value. The mode `updates` indicates that the outgoing value might be different from the incoming value, and that both are relevant. The mode `replaces` indicates that the outgoing value is set by the operation, and that the incoming value is irrelevant. The mode `clears` indicates that that the outgoing value is an initial value for its type, and that the incoming value is relevant.

3 Trivial-Update Defects

3.1 Examples

The following two examples illustrate trivial-update defects. Both exemplify actual specification errors we have made ourselves or have observed others making. The first involves a trivial typo, the second a logical mistake.

```
procedure IncreaseP (
      updates  p: Integer,
      restores max: Integer)
  requires
    p < max
  ensures
    p >= #p
```

```
procedure RemoveAny(
      updates s: Set,
      replaces x: Item)
  requires
    s /= empty_set
  ensures
    x is in #s and
    #s = s union {x}
```

Fig. 1. Specifications of `IncreaseP` and `RemoveAny`

The error in `IncreaseP` is that the post-condition should read `p > #p` (i.e., the outgoing value of `p` exceeds its incoming value). [1] Other than performing some kind of natural-language processing on the name of the operation, how can this defect be detected?

`RemoveAny` is intended to remove an arbitrary element from a non-empty set. What is the defect here, and how can it be detected?

3.2 Identifying Trivial-Update Defects

The `updates` mode is unique in that it introduces some redundancy: it summarizes the specifier's intent for the parameter at a coarse level, while the pre-condition and post-condition pin down the details. It is this redundancy that allows a trivial-update defect to be identified.

```
procedure Foo(restores r: T1, updates u: T2,
              replaces s: T3, clears c: T4)
  requires
    pre⟨r, u, c⟩
  ensures
    post⟨r, #u, u, s, #c, c⟩
```

Listing 1.1. A general operation specification schema

Consider the general schema for an operation shown in Listing 1.1 (in which the types of the parameters are unimportant). The modes determine which parameters' incoming and outgoing values may be mentioned in the pre-condition and post-condition. Indeed, conformance to this schema is one of the purely syntactic specification admissibility checks discussed in [1].

[1] A formal parameter appearing with a # prefix in an ensures clause denotes the parameter's value before the call. This is omitted in a requires clause, where all parameters necessarily denote their values before the call.

The process of identifying trivial-update defects begins by building sentence (1) claiming that there *exist* incoming values of the parameters that satisfy the pre-condition—because if not, the specification is inadmissible [1] since *any* implementation of it is correct:

$$\exists r, u, c \ (pre\langle r, u, c\rangle) \tag{1}$$

Assuming this first admissibility check is passed, we next build sentence (2) claiming that for *all* possible incoming parameter values that satisfy the pre-condition, there *exist* outgoing parameter values that satisfy the post-condition. Notice that the last two conjuncts in the post-condition are a direct result of the parameter modes, i.e., the value of `r` is restored to its incoming value, and the value of `c` is changed to an initial value for its type:

$$\forall \#r, \#u, \#c \ \ (pre\langle \#r, \#u, \#c\rangle \Rightarrow \\ \exists\ r, u, s, c \ \ (post\langle r, \#u, u, s, \#c, c\rangle \wedge (r = \#r) \wedge is\_initial(c))) \tag{2}$$

The variable r, introduced by the existential quantifier in sentence (2), is simply introducing a new name for $\#r$. Thus, it can be removed from the existential quantifier by using the name r for both:

$$\forall r, \#u, \#c \ \ (pre\langle r, \#u, \#c\rangle \Rightarrow \\ \exists\ u, s, c \ \ (post\langle r, \#u, u, s, \#c, c\rangle \wedge is\_initial(c))) \tag{3}$$

If sentence (3) is valid then we have completed another of the specification admissibility checks introduced in [1]. Assuming the specification also passes this check, we now treat the `updates`-mode parameter u as if it were `restores`-mode and build sentence (4) claiming that the contract might be implemented by code that never modifies u:

$$\forall r, \#u, \#c \ \ (pre\langle r, \#u, \#c\rangle \Rightarrow \\ \exists\ u, s, c \ \ (post\langle r, \#u, u, s, \#c, c\rangle \wedge (u = \#u) \wedge is\_initial(c))) \tag{4}$$

Finally, we can simplify this to sentence (5):

$$\forall r, u, \#c \ \ (pre\langle r, u, \#c\rangle \Rightarrow \exists\ s, c \ \ (post\langle r, u, u, s, \#c, c\rangle \wedge is\_initial(c))) \tag{5}$$

Sentence (5) is the meaning of "the specification has a trivial-update defect." So, the validity of sentence (6) (the negation of sentence (5)) indicates the admissibility of the specification with respect to trivial-update defects:

$$\exists r, u, \#c \ \ (pre\langle r, u, \#c\rangle \wedge \forall\ s, c \ \ (\neg post\langle r, u, u, s, \#c, c\rangle \vee \neg is\_initial(c)))) \tag{6}$$

3.3 Examples Revisited

For the specification of `IncreaseP`, the sentence claiming there is a trivial-update defect is clearly valid:

$$\forall p, max : integer \ (p < max \Rightarrow p \geq p) \tag{7}$$

For the specification of `RemoveAny`, the sentence claiming there is a trivial-update defect is also valid, though it is hardly as obvious:

$$\forall s : \textit{finite set of Item}\ \ (s \neq \emptyset \Rightarrow \exists\, x : \textit{Item}\ \ ((x \in s) \wedge (s = s \cup \{x\}))) \qquad (8)$$

The problem with the `RemoveAny` specification is that the second clause of the post-condition does not guarantee that x has been removed from s. A correct specification replaces this clause with `s = #s \ {x}`.

In each example, the corrected specification results in the trivial-update defect sentence being invalid.

4 *SpecChec*: A Specification Analysis Tool

We created a specification analysis tool, *SpecChec*, that tries to prove both admissibility (sentences (1), (3), (6)) and inadmissibility (their negations) of specifications. The tool is implemented inside of the Modular Verification Environment (MVE), which is the foundation for the OSU RSRG RESOLVE tools [10].

Within the MVE framework, *SpecChec* is implemented as if it were a VC generator. Once its "VCs" (i.e., the six sentences mentioned above, only half of which are true) have been created, the system first sends each sentence to the automated theorem-prover SplitDecision. If that fails to prove it, the sentence is sent to Z3. The examples discussed in Section 3 can be accessed and tested with *SpecChec* via the website `http://resolveonline.cse.ohio-state.edu/?r=NFM2014`. At this site, the examples can be found by navigating to components in the tree along the left side of the page and then opened by clicking on the name of the specification. Clicking "Verify" at the top runs the *SpecChec* tool on the specification. After this step, the VCs that were generated for a given operation can be viewed by clicking on the large dot in the left margin on the line of specification associated with that set of VCs. A red dot indicates that the specification has an admissibility problem. Yellow indicates that *SpecChec* was unable to determine admissibility.

SpecChec is often able to discharge VCs associated with the admissibility checks, described earlier, when they involve only universal quantifiers or Presburger arithmetic. However, the trivial-update defect identification technique introduces an alternation of quantifiers that automated solvers generally cannot presently handle. When automated provers become more adept at handling various mathematical types and quantifier structures, even if just for $\forall\exists$- and $\exists\forall$-style quantifier alternation, more and more specifications will be automatically checkable for admissibility.

5 Conclusions

It will never be possible to prove that the formal specification of an operation captures its informally-stated requirements. However, the use of specification modes in RESOLVE or (to some extent) `modifies` clauses in other languages,

introduces a small redundancy that provides an opportunity in principle to check whether a contract is self-consistent. The introduction of alternating quantifiers means that such admissibility checks often cannot be discharged automatically with current technology. Yet the fact that such a small degree of redundancy allows for any specification checks begs the open question of what other ways we might be able to leverage redundancy in contract specifications, e.g., perhaps by imposing other syntactic requirements for limited kinds of redundancy. JML, for example, optionally allows redundant clauses in post-conditions [8].

Acknowledgment. The authors thank all members of RSRG for their suggestions and support. This material is based upon work supported by the National Science Foundation under Grant No. CCF-1162331. Any opinions, findings, conclusions, or recommendations expressed here are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Tagore, A., Weide, B.W.: Automatically detecting inconsistencies in program specifications. In: Brat, G., Rungta, N., Venet, A. (eds.) NFM 2013. LNCS, vol. 7871, pp. 261–275. Springer, Heidelberg (2013)
2. Sitaraman, M., Adcock, B., Avigad, J., Bronish, D., Bucci, P., Frazier, D., Friedman, H., Harton, H., Heym, W., Kirschenbaum, J., Krone, J., Smith, H., Weide, B.: Building a push-button RESOLVE verifier: Progress and challenges. Formal Aspects of Computing 23(5), 607–626 (2011)
3. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
4. Adcock, B.M.: Working Towards the Verified Software Process. PhD thesis, The Ohio State University, Columbus, OH, USA (2010)
5. Westland, J.: The cost of errors in software development: evidence from industry. Journal of Systems and Software 62(1), 1–9 (2002)
6. Sitaraman, M., Weide, B.: Component-based software using RESOLVE. SIGSOFT Softw. Eng. Notes 19, 21–63 (1994)
7. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16 2010. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
8. Leavens, G.T., Leino, K.R.M., Poll, E., Ruby, C., Jacobs, B.: JML: Notations and tools supporting detailed design in Java. In: OOPSLA 2000 Companion, pp. 105–106. ACM (2000)
9. Barnett, M., Leino, K.R.M., Schulte, W.: The Spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
10. Hoffman, D.: A Framework for Integrating Automated Software Verification Tools. Tech-Report (2012), `ftp://ftp.cse.ohio-state.edu/pub/tech-report/2012/TR05.pdf`