Automatically Detecting Inconsistencies in Program Specifications

Aditi Tagore and Bruce W. Weide

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

Abstract. A verification system relies on a programmer writing mathematically precise descriptions of code. A specification that describes the behavior of an operation and a loop invariant for iterative code are examples of such mathematical formalizations. Due to human errors, logical defects may be introduced into these mathematical constructs. Techniques to detect certain logical errors in program specifications, loop invariants, and loop variants are described. Additionally, to make program specifications more concise and to make it easier to create them, RESOLVE has parameter modes: each formal parameter is annotated with a mode that is related to the intended roles of the incoming and outgoing values of that parameter. Methods to check whether the programmer has chosen a plausibly correct mode for each parameter are also explained. The techniques described are lightweight and are applied at an early stage in the verification process.

1 Introduction

The primary value of a formal verification system is to verify a program implementation against its specification and to report an implementation error if there is one. The robustness of such a system depends as much on the programmer supplying a correct specification for her program as it does on the theorem prover's ability to prove the verification conditions (VCs) generated from a proposed implementation of that specification. However, inconsistencies in the specification may be introduced during the software development process due to human errors. In such scenarios, either an implementation may be declared as correct for an incorrect specification, or it may not be possible to write a valid implementation at all. In a similar way, defects may occur when a programmer annotates a loop with an invariant and a variant.

The idea described in this paper is used to detect certain errors at an early stage in the formal verification process. Typically, errors are only detected in a verification system when a VC cannot be proved by the theorem prover and subsequently the VC is traced back to its origin in the program to identify the error. We describe a lightweight method that checks consistency of certain programming constructs before VCs are generated. Since the cost of detecting

G. Brat, N. Rungta, and A. Venet (Eds.): NFM 2013, LNCS 7871, pp. 261–275, 2013.

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.

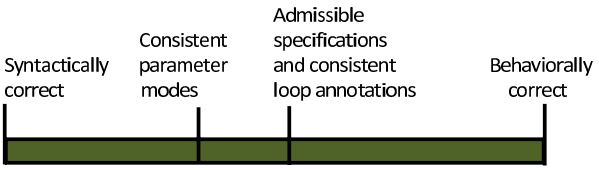

Fig. 1. Types of correctness checks

Formal verification ensures that a program is behaviorally correct, i.e., it matches its specification. It is far stonger than syntactic correctness of a program, which is checked by an ordinary compiler. The techniques outlined here lie between these two extremes. The consistency of program specifications and loop annotations is accomplished with the help of a theorem prover (also used for formal verification), but instead of proving an entire program to be correct, we perform local checks on mathematical statements that do not depend on the entire body of code. Additionally, we also illustrate methods to accomplish consistency checks on the modes of operation parameters. These are syntactic checks, but of a slightly different order than those that are normally carried out by a compiler. This classification is shown in Figure 1.

Our specifications and their implementations are written in RESOLVE [1]. To detect logical inconsistencies in program specifications and their implementations, we use an SMT solver, Z3 [2], as a back-end prover. The example programs are chosen from the RSRG software components library, some of which have been suggested as software verification benchmarks [3]. Students in computer science classes have been observed to make the kinds of errors that are mentioned in this paper, as have the authors and other more experienced specifiers.

The contributions in this paper are three-fold:

- The conditions for admissibility of program specifications are formulated.
- Techniques to establish logical consistency of loop annotations (invariants and variants) are developed.
- Methods for ascertaining that a programmer has supplied the correct modes for the parameters of an operation are described.

A reader's familiarity with formal specifications, pre- and post-conditions, and loop invariants and variants is assumed. However, no prior knowledge of RESOLVE or of the intricate mechanisms of Z3 is necessary. The ideas apply to specifications and formal verification / theorem-proving technology in general.

Section 2 provides an overview of the types of defects considered. Sections 3,4 and 5 expound on the techniques for detecting such defects with examples. Discussion and related work are presented in Section 6, with conclusions in Section 7.

2 Types of Specification Defects Detected

2.1 Defective Contracts

An operation (or method) specification promises certain properties that the implementer can assume at the time of a call via the `requires` clause (pre-condition) and, in turn, demands that certain properties hold upon return via the `ensures` clause (post-condition). We first divide specifications into two distinct groups: those that are implementable and those that are not. Unimplementable specifications are, for our (practical) purposes, considered inadmissible. A specification with an unsatisfiable post-condition is unimplementable and hence inadmissible. On the other hand, not all specifications that are implementable are admissible[1]. Some of them may be trivially correct and hence are inadmissible. This happens if the pre-condition is unsatisfiable. This characterization is shown in Figure 2. In short, an inadmissible specification is one that, for practical purposes, must have resulted from a specification error. In Section 3 we discuss various techniques to detect these different types of problems in specifications.

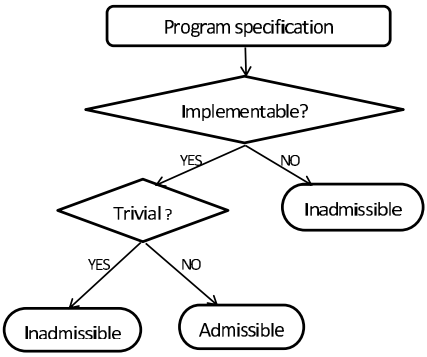

Fig. 2. Program Specifications

2.2 Defective Loop Annotations

Loop invariants and variants are important constructs needed to formally verify a program. Loop invariants are needed to reason about the loop, without considering the loop iterations individually. Correspondingly, to prove the total correctness of programs, variants (called progress metrics in RESOLVE) are used. A variant is usually a natural number that has a positive value before each time the loop body is executed, and must be reduced in each iteration. Invariants and variants are together used to prove total correctness of loops.

As mentioned earlier, there is a benefit to detecting defects in loop annotations at an early stage, even before VCs are generated. In Section 4 we discuss techniques to ensure admissibility of its constituent parts.

2.3 Inconsistent Parameter Modes

Each parameter of a RESOLVE operation is annotated with a *mode* in the header of that operation. A programmer may fail to select parameter modes that

[1] "Admissible" refers to a specification that does not contain any checkable defects, not to one that "correctly formalizes the requirements".

are consistent with how the parameter values are utilized and changed in the operation. In other words, the modes may not be consistent with the `requires` and the `ensures` clause.

For example, the `replaces` mode indicates that the outgoing parameter value is determined by the operation and that the incoming value is inconsequential. Thus the operation should not refer to the incoming value of the parameter. Hence if the programmer uses a `replaces` mode when the incoming value of the parameter appears in the pre- or post-condition, then she is alerted of this anomaly.

A detailed explanation of the modes is presented in Section 5 where we also discuss the methods we employ to detect modes that are inconsistent with the specification of the operation.

3 Inconsistent Specifications

3.1 Methodology

A specification may be inadmissible for various reasons, as discussed in Section 2.1. We perform a series of checks to identify those that are not admissible. The following lists a taxonomy of defects that may occur in the specifications.

Contradiction in the Pre-condition: An implementation may be declared as trivially correct if the pre-condition (say *pre*) is $false$. Logical contradictions appearing in *pre* make it $false$ and hence the post-condition (say *post*) is irrelevant. To avoid such *default* correctness, the `requires` clause is tested for satisfiability.

Contradiction in the Post-condition: On the other hand, a specification is unimplementable if *post* is $false$. A contradiction in *post* implies that it is impossible to create an implementation that meets the specification. In this case too, the `ensures` clause is tested for satisfiability.

Appropriateness of *pre* and *post* Together: Even when the `requires` and `ensures` clauses are individually contradiction-free, they may still preclude an implementation that satisfies the program specification.

```
procedure DecrementBy3 (updates x: Integer)
 requires
   x >= 3
 ensures
   x = #x - 3 and x > 0
```

Fig. 3. Proposed specification of `DecrementBy3`

As an example, consider the specification of a procedure `DecrementBy3` shown in Figure 3. The parameter mode `updates` for the parameter `x` indicates the value of this parameter may be changed by the operation. The `#` symbol in

the `ensures` clause refers to the parameter at the time of the call; no `#` symbol is used in the `requires` clause as it always refers to the incoming value. The pre-condition of the procedure `DecrementBy3` says that the incoming value of `x` should be greater than or equal to 3. The post-condition of the procedure says that the outgoing value of `x` is equal to 3 less than its incoming value (denoted by `#x`) *and* that the outgoing value is positive.

The `requires` and `ensures` clauses are individually satisfiable: both are *true* for `#x = 4` and `x = 1`. Even so, a valid implementation (that is correct for all the input values that satisfy the pre-condition) is still not possible, because the input value `#x = 3` satisfies the `requires` clause, but the outgoing value of `x` must be 0 and that makes the second conjunct `(x > 0)` $false$, and hence invalidates the `ensures` clause.

To detect logical inconsistencies in program specifications (e.g., the one in Figure 3), we need to ascertain that for *all* possible values that can satisfy *pre*, there *exist* values of the variables that satisfy *post*. So we check the validity of

$$\forall x_1, \ldots, x_n (pre \implies \exists y_1, \ldots, y_m (post)) \tag{1}$$

where $(x_1, \ldots, x_n)$ are the incoming values of the variables appearing in *pre* and $(y_1, \ldots, y_m)$ are the outgoing values of the variables in *post*.

Thus the specification in Figure 3 is tested with the help of the formula

$$\forall \#x (\#x >= 3 \implies \exists x (x = \#x - 3 \wedge x > 0)) \tag{2}$$

3.2 Example: A Divide Operation for Unbounded Integers

In RESOLVE, a `contract` contains the client-view of a software component that describes a model of that component's behavior. A `realization` module contains operation bodies that implement the operations specified in the contract. Consider the contract `Divide` for *unbounded* integers, i.e, integer values without an upper or lower bound.

```
contract Divide enhances UnboundedIntegerFacility

 procedure Divide (updates i: Integer, restores j: Integer,
                                        replaces r: Integer)
    ensures
        #i = i * j + r  and  0 < r  and  r < |j|

end Divide
```

Fig. 4. Proposed specification of `Divide`

In the `Divide` operation (shown in Figure 4), the incoming value of `i` (denoted by `#i`) is the dividend and the quotient is the outgoing value of `i`. Since the value of the divisor `j` remains unchanged, its parameter mode is `restores`.

The remainder from the division is returned in `r`. Since, the incoming value of `r` is inconsequential, its parameter mode is `replaces`.

An important observation needs to be made about the variable `j`. The parameter mode of `j` is `restores`, which means the incoming and the outgoing values are the same; it is equivalent to having `j = #j` as part of the `ensures` clause. For simplicity, a programmer can leave such a clause out of the post-condition. But while constructing the formula to check for validity, this additional conjunct needs to be appended to the `ensures` clause. First, we check whether the pre-condition (there isn't one and hence by default is it considered to be *true*) and post-condition are individually satisfiable; they are. Then, we construct formula (3) to check for the admissibility of the specification.

$$\forall \#i, \#j(true \implies \exists i, j, r((\#i = i * j + r) \wedge (0 < r) \wedge (r < |j|) \wedge (j = \#j))) \tag{3}$$

To test whether the specification is admissible, formula (3) is automatically translated into Z3's SMT2 input format and Z3 is invoked to prove it. Z3 determines that it is invalid. This gives a flag to the programmer that the specification contains an error. When a formula is determined to be invalid, Z3 produces a counter-example, i.e., values for which the formula does not hold. Here, it suggests a value 0 for `#j`. As we know that `j = #j`, the last two conjucts of the `ensures` clause in Figure 4 are reduced to `0 < r` and `r < 0`. Since a conflict arises, the programmer (we hope) sees that a value of 0 cannot be allowed for the divisor `j`. Hence, this should be prevented by the `requires` clause.

```
requires
 j /= 0
ensures
 #i = i * j + r  and  0 < r  and  r < |j|
```

Fig. 5. New proposed pre- and post-condition of `Divide`

The specification in Figure 4 is now corrected as shown in Figure 5, and it is checked again:

$$\forall \#i, \#j((\#j \neq 0) \implies \exists i, j, r((\#i = i{*}j{+}r){\wedge}(0 < r){\wedge}(r < |j|){\wedge}(j = \#j))) \tag{4}$$

Z3 declares this formula invalid as well and produces a counter-example where `#j` = 1. On substituting this value, the last two conjucts of the ensures clause are `0 < r` and `r < 1`. The variable `r` cannot satisfy both these conjuncts at the same time, since `r` is an integer. Hence, the programmer should realize at this point, that the remainder from the `Divide` operation may be equal to 0.

```
requires
  j /= 0
ensures
  #i = i * j + r  and  0 <= r  and  r < |j|
```

Fig. 6. Correct Pre- and post-condition of `Divide`

Hence, the specification for `Divide` is updated one last time to the admissible one in Figure 6.

3.3 Example: An Increment Operation for Bounded Integers

We next consider a contract for *bounded* integers, where two constants `MIN` and `MAX` represent the minimum and the maximum bounds respectively. The bounds are used with the restriction that `MIN <= 0` and `0 < MAX`. Figure 7 shows (some of) a proposed contract for the `BoundedIntegerFacility`.

```
contract
 BoundedIntegerFacility

 definition MIN: integer
  satisfies restriction
   MIN <= 0

 definition MAX: integer
  satisfies restriction
   0 < MAX

 math subtype INTEGERMODEL
  is integer
  exemplar i
  constraint
   MIN <= i  and  i <= MAX

type Integer is modeled
   by INTEGERMODEL
 exemplar i
  initialization ensures
  i = 0

procedure Increment
 (updates i: Integer)
 requires
  i <= MAX
 ensures
  i = #i + 1
 ...
end BoundedIntegerFacility
```

Fig. 7. A proposed `BoundedIntegerFacility` contract

The specification of each operation that appears in this contract can be tested for correctness using the method described in Section 3.1. However, a bounded integer (say `i`) in this contract must satisfy `MIN <= i < MAX` that is introduced by the `constraint` clause. Thus, to check the validity of each of the operation specifications in this contract, formula (1) needs to be updated such that the `constraint` clause (say *constr*) on the program variables (an abstract invariant) is not violated. The `constraint` clause must hold for both the incoming and the outgoing parameter values and thus needs to be appended to

both the `requires` and the `ensures` clauses. In addition, the `restriction` clause (say *restr*) on the boundary values (i.e., `MAX` and `MIN`) must also be accounted for in the formula in the same way as the `constraint`. The resultant formula is shown in (5).

$$\forall x_1, \ldots, x_n((pre \wedge constr \wedge restr) \implies \exists y_1, \ldots, y_m(post \wedge constr \wedge restr)) \tag{5}$$

where $(x_1, \ldots, x_n)$ are the incoming values of the parameters appearing in *pre*, *constr* and *restr* and $(y_1, \ldots, y_m)$ are the outgoing values of the parameters in *post*, *constr* and *restr* .

The specification of the operation `Increment`, like all others, needs to be tested for validity using the formula from (5). Substituting values, the formula evaluates to

$$\begin{aligned}\forall \#i, MAX, MIN(((\#i <= MAX) \wedge (\#i <= MAX) \wedge (MIN <= \#i) \\ \wedge (0 < MAX) \wedge (MIN <= 0)) \implies \exists i((i = \#i + 1) \wedge (i <= MAX) \\ \wedge (MIN <= i) \wedge (0 < MAX) \wedge (MIN <= 0)))\end{aligned} \tag{6}$$

Z3 concludes that the formula in (6) is invalid, and produces a counter-example with a value 1 for each of the variables `#i` and `MAX`. Substituting these values in the formula (6), the programmer notices that the `ensures` clause no longer holds true, as the value of `i` becomes greater than `MAX`. This gives the programmer a clue that to keep the value of `i` within bounds, the value of `#i` should have been less than `MAX`.

On correcting the `requires` clause of the specification, the resulting specification of `Increment` is tested for validity and Z3 determines it to be valid.

3.4 Example: A Halve Operation

Like many other languages for writing specifications, RESOLVE supports user-defined mathematical functions and predicates. The procedure `Halve` in Figure 9 contains a user-defined predicate `IS_ODD`, that is presented in Figure 8. The ability to make up new definitions helps the specifier: instead of writing out the expression for *odd* each time, she can condense it with the help of the predicate. This also can help the prover [4].

To test the admissibility of the `Halve` specification, the formula to be checked for validity is in (7).

$$\begin{aligned}\forall \#i(true \implies \exists i((IS\_ODD(\#i) \implies \#i = i + i) \\ \wedge (\neg IS\_ODD(\#i) \implies \#i = i + i + 1)))\end{aligned} \tag{7}$$

Z3 declares this as invalid and produces a value 0 for each of the variables `#i` and `i`. Since 0 does not satisfy the `IS_ODD` predicate, substituting the values gives rise to the expression $0 = 0 + 0 + 1$, which is impossible. At this point the programmer realizes that the conditions are simply flipped and she updates the specification to the correct one. The new check of the specification indicates that it is admissible.

```
definition IS_ODD(i: Integer)
        : boolean
 is
    i mod 2 /= 0
```

Fig. 8. The predicate *IS_ODD*

```
procedure Halve(
    updates i: Integer)
  ensures
   if IS_ODD(#i) then
    #i = i + i
   else
    #i = i + i + 1
```

Fig. 9. Proposed specification for `Halve`

4 Consistency of Loop Annotations

In RESOLVE, a loop invariant for a `while` loop is introduced via a `maintains` clause. This clause formalizes the relation between the variable values just before the loop (prefixed with a `#`) and the variable values at any time the while loop condition is checked (unadorned). Additionally, the progress metric (variant) of a loop is stated in a `decreases` clause.

```
procedure Add (updates n: Natural, restores m: Natural)
 variable k, z: Natural
 loop
  maintains n + m = #n + #m  and  k + m = #k + #m  and  z = 0
  decreases m
 while not AreEqual (m, z) do
  Increment (n)
  Increment (k)
  Decrement (m)
 end loop
 m :=: k
end Add
```

Fig. 10. Procedure `Add` for `UnboundedNaturalFacility`

Figure 10 shows the code for operation `Add` for natural numbers. This procedure adds two natural numbers `n` and `m` and stores the result in `n`. The body of this procedure makes calls to two other operations: `Increment` and `Decrement`, which have been defined in the `UnboundedNaturalFacility` contract. The primary data-movement operator in RESOLVE, :=:, swaps (exchanges) the values of its two operands, which must be simple variables. We perform the following checks to ensure that the constituent parts of a loop are admissible.

4.1 The Invariant and the Boolean Condition Are Contradiction-Free

If two or more conjuncts in the loop invariant (say *inv*) contradict each other, then the invariant evaluates to *false* and the loop will never execute. For a

similar reason, the boolean loop condition (say B) should not contain any contradictions. Thus we first check to see that inv and B are individually satisfiable.

4.2 The Variant Is Positive Every Time the Loop Executes

In order for a loop to execute, the loop variant (say var) must (a) be positive every time the loop body executes, and (b) decrease during every iteration of the loop (we restrict attention to loop variants that are non-negative integers). We perform consistency checks to see that case (a) holds. Since to prove the validity of (b), the loop body needs to be involved, this is generated as a VC later in the tool chain.

To ensure that case (a) holds, we check the validity of the following formula.

$$\forall x_1, \ldots, x_n (B \wedge inv \implies var > 0) \tag{8}$$

where $(x_1, \ldots, x_n)$ are the variables that occur in the boolean condition, the invariant and the variant. Applying this formula to check the validity of the variant in Figure 10, the formula to be tested for validity is

$$\forall m, \#m, n, \#n, k, \#k, z (m \neq z \wedge n + m = \#n + \#m \wedge k + m = \#k + \#m \wedge z = 0 \implies m > 0) \tag{9}$$

Here, since m and z are natural numbers, their values have to be at least 0 and thus, the above formula is valid.

4.3 The Loop Invariant Is Valid Before the Loop Executes for the First Time

Some fundamental properties of a loop invariant are that it holds (a) the first time before entering the loop, and (b) at the end of each iteration of the loop. As in the case of variants, to prove case (b), the loop body needs to be examined, and thus a VC is generated later for this purpose. Here for case (a), we perform a simple check of the logical consistency of the loop invariant before entering the loop. We need to ensure that there *exist* some values of variables in the invariant such that it is potentially true. Thus we check the validity of (10).

$$\exists x_1, \ldots, x_n (inv_{init}) \tag{10}$$

where inv_{init} is the invariant with # symbols removed and $(x_1, \ldots, x_n)$ are the variables in the invariant. By definition of #, before the loop executes for the first time, each unadorned variable in the invariant has the same value as the adorned version.

Application of this procedure to the invariant in Figure 10 results in

$$\exists k, z, n, m (n + m = n + m \wedge k + m = k + m \wedge z = 0) \tag{11}$$

The validity of formula (11) confirms that the loop invariant *might* be valid at the beginning of the loop. A VC is generated later in the tool-chain to see whether it is *always* valid at this point.

5 Detecting Incorrect Parameter Modes

RESOLVE has multiple parameter modes: `restores`, `updates`, `replaces` and `clears`. Although some of them have been mentioned in previous sections, their meanings are consolidated in Table 1.

Table 1. Parameter modes

Parameter Mode	Description
`restores`	The incoming and the outgoing values of the parameter are the same
`updates`	The incoming and outgoing values of the parameter are potentially different
`replaces`	The operation's behavior does not depend on the incoming value (a special case of `updates`)
`clears`	The outgoing value of the parameter is an initial value of its type (a special case of `updates`)

The programmer supplies a mode for each parameter, as discussed in Section 2.3. In our technique, syntactic checks are employed to give suggestions to the programmer about the appropriate mode in case the way in which the parameter values in an operation are utilized are not consistent with the parameter mode. In our method, we allow that there might be an error either in the pre- or post-condition or in the parameter mode. In other words, we do not assume the parameter mode or the body of the specification to be absolutely correct; instead we make suggestions to assist the programmer write a correct specification when these are not consistent with each other.

5.1 A Variable with `Replaces` Mode Appears in the `Requires` Clause

If a variable occurs in the `requires` clause, the incoming value of the variable is relevant to the operation's behavior. But this cannot occur if the mode is `replaces`. Thus if the incoming value of a variable annotated with the `replaces` parameter mode is used in the `requires` or `ensures` clause, the programmer is issued a warning message.

```
procedure Divide (updates i: Integer, replaces j: Integer,
                                        replaces r: Integer)
  requires
        j /= 0
...
```

Fig. 11. Incorrect header of `Divide`

As an example, consider the `Divide` procedure from Section 3.2. The value of the divisor `j` remains unchanged, and thus the parameter mode should correctly be `restores`. Suppose instead, as shown in Figure 11, if the programmer incorrectly uses the `replaces` mode, an error is detected, since the incoming value of j is used in the `requires` clause to state that division by 0 is not allowed.

5.2 Incoming Value of a Variable in the Post-condition

The incoming value of a variable (say x) may occur in the `ensures` clause *if and only if* the parameter mode is `updates` or `clears`. For the `restores` mode, since the incoming and outgoing values of the variables are the same, $\#x$ should not appear in the `ensures` clause (because x should be used). Although the value of the variable is changed with `replaces` mode, the `ensures` clause cannot refer to the incoming value of the variable, since the incoming value is supposed to be immaterial.

Thus, a warning is given to the programmer in the following two cases:

- She used the parameter mode `updates` or `clears` and yet did not use $\#x$ in the post-condition. This could mean that either she wanted the mode to be either `restores` or `replaces` or that she missed out an additional conjunct in the `ensures` clause that refers to the incoming value of the variable.
- She designated the mode to be `restores` or `replaces` and $\#x$ appeared in the `ensures` clause.

```
procedure Add
 (updates n: Natural,
  restores m: Natural)
```

(a) The header of `Add`

```
ensures
    n = n + m
```

(b) Incorrect post-condition

```
ensures
    n = #n + #m
```

(c) Incorrect post-condition

```
ensures
    n = #n + m
```

(d) Correct post-condition

Fig. 12. The `Add` operation

Figure 12 shows the contract for `Add` for natural numbers. The body of `Add` operation was illustrated in Figure 10. The header for this procedure is shown in (a). If the programmer writes the post-condition as shown in (b), an error is detected since the parameter mode for `n` is declared as `updates`, yet the incoming value `#n` does not appear in the post-condition. On the other hand, if

the programmer writes the post-condition as in (c), an error is detected again since the mode for `m` is `restores`, and yet the incoming value `#m` appears. The correct post-condition is shown in (d).

5.3 Other Warnings

In addition to the errors listed above, warnings are given to the programmer when the following anomalies are noticed:

- **The `clears` parameter mode**: When the parameter mode is `clears`, typically the outgoing parameter value is not referenced in the post-condition since the parameter value is reset to an initial value of its type.

  For example, suppose a programmer declares the parameter mode of `i` to be `clears`, and then adds a conjunct to the post-condition which says `i = 0`. This conjunct is unecessary, since the RESOLVE compiler automatically adds the conjunct `i = 0` to the post-condition when the parameter mode of `i` is `clears`.

- **The outgoing parameter equals a constant value in the post-condition**: If the mode of the variable is `restores`, then a conjunct `i = constant` should not appear in the post-condition. Since the parameter value is not changed by the operation, it is not meaningful to add a conjunct stating that the outgoing value should be equal to a particular value. (It is okay, of course, to say that some other variable is equal to this parameter.)

6 Discussion and Related Work

The examples of inconsistent specifications that are presented in Section 3 concern integers. The method presented here to detect such inconsistencies can also be applied for other datatypes such as arrays, stacks, queues, etc. However, Z3 is frequently unable to determine a formula as valid / invalid if it contains recursive definitions of datatypes. In addition, our primary admissibility-check formula contains an alternation of quantifiers that automated solvers have trouble with. It is expected that if provers become more adept at handling such datatypes and quantifiers, a larger range of specifications can be automatically checked for admissibility.

Validating program specifications has been previously described as generating test-cases [5] and as symbolic execution [6, 7]. Both of these methods rely on making the specifications executable. However, formal specifications are non-executable mathematical statements, and to conform with this characteristic, our technique uses a theorem prover to establish their validity. Heitmeyer, *et.al* [8] describe a toolset to carry out syntactic and critical property checks like safety, timing, etc., on specifications. In contrast, the analyses in this paper are more general in that they do not depend only on certain properties of particular types of specifications, but check for logical consistency of *every* program specification. They are also more involved than mere syntactic checks (as depicted in Figure 1).

Some program verifiers such as Dafny [9] are capable of detecting a subset of inconsistent specifications that are described in this paper. Dafny is capable of detecting a "division by zero" error in the `ensures` clause that states the return value as `i/j`, i.e., it explicitly uses the divide operator, when a `requires` clause stating that `j` $\neq$ `0` is missing. But, it does not detect a more involved division by zero error such as the one present in the `Divide` specification shown in Figure 4.

Ponsini *et.al* [10] describe a way of determining the correctness of loop invariants using constraint solvers. Their correctness proof closely follows Hoare logic [11]. The admissibility checks presented in this paper are of a different nature and are more comprehensive in that consistency of loop variants is also considered.

7 Conclusions

In this paper, we developed methods to detect logical inconsistencies in program specifications and errors in loop invariants and variants. Methods to help the programmer annotate each parameter in operation headers with the correct mode are also presented.

Most inconsistencies that are identified by our techniques are logical ones: those that if present might cause an error during verification. By detecting them early, we prevent the programmer from making more mistakes further along. However, some of the inconsistencies (mentioned in Section 5.3) are warnings and not errors, in that they do not cause the verification process to fail and yet are better eliminated so that a better program specification is achieved.

In the future, we will contine to enhance our technique to detect logical inconsistencies in other specification constructs. We hope that others also implement and extend this idea to create useful lightweight tools that help programmers by leveraging formal specifications.

Acknowledgment. The authors are grateful for the suggestions of the members of RSRG. This material is based upon work supported by the National Science Foundation under Grants No. CCF-0811737, ECCS-0931669, and 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. Sitaraman, M., Weide, B.: Component-based software using RESOLVE. SIGSOFT Softw. Eng. Notes 19, 21–63 (1994), http://doi.acm.org/10.1145/190679.199221
2. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)

3. Weide, B.W., et al.: Incremental benchmarks for software verification tools and techniques. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 84–98. Springer, Heidelberg (2008)
4. Tagore, A., Zaccai, D., Weide, B.W.: Automatically proving thousands of verification conditions using an SMT solver: An empirical study. In: Goodloe, A.E., Person, S. (eds.) NFM 2012. LNCS, vol. 7226, pp. 195–209. Springer, Heidelberg (2012)
5. Kemmerer, R.: Testing formal specifications to detect design errors. IEEE Transactions Software Engineering, 32–43 (1985)
6. Kneuper, R.: Symbolic execution as a tool for validation of specifications. PhD thesis, University of Manchester (1989)
7. Bouquet, F., Dadeau, F., Legeard, B., Utting, M.: Symbolic animation of JML specifications. In: Fitzgerald, J.S., Hayes, I.J., Tarlecki, A. (eds.) FM 2005. LNCS, vol. 3582, pp. 75–90. Springer, Heidelberg (2005)
8. Heitmeyer, C., Kirby, J., Labaw, B., Bharadwaj, R.: SCR: A toolset for specifying and analyzing software requirements. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 526–531. Springer, Heidelberg (1998)
9. Leino, K.R.M.: Dafny: An automatic program verifier for functional correctness. In: Clarke, E.M., Voronkov, A. (eds.) LPAR-16. LNCS, vol. 6355, pp. 348–370. Springer, Heidelberg (2010)
10. Ponsini, O., Collavizza, H., Fedele, C., Michel, C., Rueher, M.: Automatic verification of loop invariants. In: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM 2010, pp. 1–5. IEEE Computer Society, Washington, DC (2010)
11. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM 12(10), 576–580 (1969)