# Holistic Specifications for Robust Programs

– getting to the point –

AUHTOR, Place

Functional specifications of program components describe what the code *can* do — the *sufficient* conditions to invoke the component's operations: a client who supplies arguments meeting that operation's preconditions can invoke it, and obtain the associated effect. While specifications of sufficient conditions may be enough to reason about complete, unchanging programs, they cannot support reasoning about components that interact with external components of possibly unknown provenance. In this *open world* setting, ensuring that your component is robust even when executing with buggy or malicious external code is critical. *Holistic specifications* — as their name implies — are concerned with the *overall* behaviour of a component, in all possible interleavings of calls to the component's operations with those of the external code. Thus, holistic specifications are concerned with *sufficient* conditions, what is enough to *cause* some effect, as well as with *necessary* conditions, what are the conditions without which an effect will not happen. By adopting holistic specification techniques, programmers can explicitly define what their components should not do, making it easier to write robust and reliable programs.

In this paper we argue for the need for such holistic specifications, propose a language for writing specifications, give a formal semantics, and discuss several examples from the literature.

## 1 INTRODUCTION

Software guards our secrets, our money, our intellectual property, our reputation [**?**]. We entrust personal and corporate information to software which works in an *open* world, where it interacts with third party software of unknown provenance, possibly buggy and potentially malicious.

This means we need our software to be *robust*. We expect software to behave correctly even if used by erroneous or malicious third parties. We expect that our bank will only make payments from our account if instructed by us, or by somebody we have authorised, that space on a web given to an advertiser will not be used to obtain access to our bank details [**?**], or that an airline will not sell more tickets than the number of seats. The importance of robustness has led to the design of many programming language mechanisms to help developers write robust programs: constant fields, private methods, ownership [**?**] as well as the object capability paradigm [**?**], and its adoption in web systems [**???**], and programming languages such as Newspeak [**?**], Dart [**?**], Grace [**??**], and Wyvern [**?**].

While such programming language mechanisms make it *possible* to write robust programs, they cannot *ensure* that programs are robust. Ensuring robustness is difficult because it means different things for different systems: perhaps that critical operations should only be invoked with the requisite authority; perhaps that sensitive personal information should not be leaked; or perhaps that resource belonging to one user should not be consumed by another. To be able to ensure robustness, we need ways to specify what robustness means for a particular program, and ways to demonstrate that the particular program adheres to its specific robustness requirements.

Consider the code snippets from Figure 1. It is about objects of class Wallet, which hold a balance and a secret, and where only the holder of the secret can use the wallet to make payments – for the sake of simplicity, we allow balances to grow negative. We show the code in three versions; each of these versions has the same method pay, and the two last versions have an additional method sendSecret. We use a Java-like syntax, and assume an untyped language (as we are in the open world setting). Thus, the classical Hoare triple describing the behaviour of pay would be:

Author's address: AuhtorPlace.

–. 2475-1421/–/13-ART $15.00
https://doi.org/

```
class Wallet{
  fld balance
  fld secret
  mthd pay(who,amt,scr){
    if (secret==scr)&amnt>0 then
      balance-=amnt
      who.balance+=amt    }
}
```

```
class Wallet{
  fld balance
  fld secret
  mthd pay(...){
    ...as version 1...  }
  mthd setSecret(secr){
      secret=secr }
}
```

```
class Wallet{
  fld balance
  fld secret
  fld owner
  mthd pay(...){
    ...as version 1...  }
  mthd sendSecret( ){
    owner.take(secret) }
}
```

Fig. 1. Three Versions of the class Wallet

```
method pay(who,amt,scr)
PRE:  this,who:Wallet ∧ this≠who ∧ amt:ℕ ∧ scr=secret
POST: this.balance=this.balance-amt ∧ who.balance=who.balance+amt
```

The above specification shows that knowledge of the secret is a *sufficient* condition to make payments. But it does not show that it is a *necessary* condition. To make the specification more "robust" we can also describe the behaviour of unction pay when the pre-condition is not satisfied:

```
method pay(who,amt,scr)
PRE:  this:Wallet ∧ ¬ (this≠who ∧ amt:ℕ ∧ this.scr=secret )
POST: ∀w: Wallet. w.balance=w.balance-amt
```

The specification from above mandates that the method pay cannot make a payment unless the secret is provided. But it cannot preclude that Wallet – or some other class, for that matter – contains more methods which might make it possible to affect a reduction in the balance without knowledge of the secret. To preclude this, we introduce *holistic specifications*, and require that:

$$\text{(Spec1)} \triangleq \forall w, m.[\ w : Wallet \wedge w.balance = m \ \wedge w.balance < m \ \longrightarrow \exists o.[\ o \wedge ow.secret\ ]\ ]$$

(Spec1) mandates that for any wallet w defined in the current configuration, if some time in the future the balance of w were to decrease, then at least one external object (an object whose class is not Wallet) in the current configuration has direct access to the secret. This external object need not have caused the change in $w.balance$ but it would have (transitively) passed access to the secret which ultimately did cause the change in the balance.

The class Wallet from Figure 1.version 1, satisfies (Spec1), but Wallet from Figure 1.version 2, does not. It is possible to overwrite the secret of the Wallet and then to effect a payment. Neither does Wallet from Figure 1.version 3, sarisfy (Spec1), since it is possible for the owner not to know the secret and the secret to be communicated to them. Insteed, the class saisfies (Spec2) from below

$$\text{(Spec2)} \triangleq \forall w, m.[\ w : Wallet \wedge w.balance = m \ \wedge w.balance < m \ \longrightarrow \exists o.[\ o \ \wedge\ (\ ow.secret \ \vee o = w.owner\ )\ ]$$

(Spec2) mandates that for any wallet w defined in the current configuration, if some time in the future the balance of w were to decrease, then at least one external object in the current configuration has direct access to the secret, or is the owner of the Wallet. The class Wallet from Figure 1.version 1 and version 3 satisfy (Spec2), but Wallet from Figure 1.version 2, does not.

## 2 CONCLUSION

zzzzz