Assertion-based Testing of Go Programs
Master's Thesis Project Description

Eva Charlotte Mayer

Supervisors:
Linard Arquint, Felix Wolf, Prof. Dr. Peter Müller (ETHZ)
and Prof. Dr. Alexander Pretschner (TUM)

May 1st 2020 – October 31st 2020

Introduction

Runtime assertions are useful for identifying bugs in programs and thus increase confidence in the program's correctness. Runtime assertions are predicates that reason about a given program's state at some point of the execution. If an assertion does not hold, the program is deemed to be in a faulty state. In comparison to unit tests that consider only function outputs, assertion-based tests allow for checks on arbitrary points in the program. In the right circumstances, this enables a programer to find the sources of errors faster than with unit tests.

Guarantees obtained through assertion-based testing reason only about a given program execution. Conversely, formal verification provides guarantees, which are even stronger because they hold for all inputs, thread schedules, and program environments. The verification tool Gobra [2] proves correctness of programs written in Go [15]. In our work we develop an assertion-based testing framework based on the same specification language used by Gobra. We do not aim to replace deductive verification, but instead supplement it. Concretely, we aim to solve the following challenges:

- Verification requires very thorough specifications which are hard to write for programers without experience in formal methods. Specifications for assertion-based testing are typically simpler as they reason about specific runtime values and also give weaker guarantees by only specifying certain properties. E.g. permissions are normally not used and also aliasing is often not expressed. Even though they are easier to formulate, they can still provide a valid basis for specification used during verification. Therefore, we intend to use the same formal language to express Gobra specifications and Go assertions. The goal is to reuse some of the runtime specifications for later verification.
- An unsuccessful assertion check at runtime provides an immediate counter example, which can be used for inspection of both the program and the

specification. Hence, rigorous testing before applying verification not only decreases the likelihood of bugs but also gives programmers confidence in the specification. After testing, if verification fails against a specification that reuses some of the runtime assertions, a user can be more certain that either the program is erroneous or the specification does not yet express strong enough properties for formal verification, than that verification is based on a faulty specification.

In summary, assertion-based testing of Go programs is a helpful supplement to verification with Gobra. The goal of this thesis is to provide such a testing framework, which motivates Go programmers to write assertions that enable synergies with the Gobra verifier. The thesis thus contributes to bridging the gap between programming and verification.

Preliminaries

Different ways to write the specification of a program exist. The approaches we will deal with in this thesis are pre- and postconditions, `assert`-statements and invariants. A precondition is a predicate that should evaluate to true when entering a function. It formulates constraints on the function's input parameters or the program state before executing the function. A postcondition is a predicate that should hold after exiting a function. It expresses the conditions that a function should ensure for its return values and the resulting program state. An `assert`-statement is a predicate that can be checked at any program point and reasons about the current program state. Two types of invariants need to be distinguished: class and loop invariants. A class invariant is a condition that all objects of a class must satisfy at every program point that can be observed from outside the class. A loop invariant is a condition that is true at the beginning and end of every loop iteration. Note that it also has to hold if the loop body is executed zero times. We will only consider loop invariants in this thesis.

Finally, it is important to discuss static versus dynamic verification: Static verification refers to verification of a program without it being executed. Using static verification, program correctness can be established. Dynamic verification checks program correctness during its execution. It finds errors at runtime. In our case, Gobra performs static verification whereas the assertion-based testing framework that is designed in this thesis accomplishes dynamic verification.

Related Work

The term "assertion" was first introduced by Goldstine's and von Neumann's work on reasoning about programs [8]. Later, assertions were included directly into programming languages with Eiffel [19] being the first object-oriented language "to support and strongly advocate the use of assertions" [8]. Eiffel is most known for implementing the concept of design-by-contract through invariants, pre- and postconditions. Many programming languages have adopted the design-by-contract paradigm and provide extensive runtime assertion testing frameworks that check invariants and pre- and postconditions at runtime, some of which can be joined with program verifiers. SPARK [3] defines a subset of the Ada programming language and includes code contracts that support both dynamic and static verification. Spec# [4] extends C# with a specification

language and a verifier. For Spec#, efforts have been made to supply programmers with meaningful counter examples when verification using Spec# fails [20]. Specification for Java programs is often written in the Java Modeling Language (JML) [17]. A variety of tools that rely on JML test or verify Java programs. Examples include, but are not limited to, the jmlc compiler [18] and the Extended Static Checker for Java (ESC/Java2) [9]. The jmlc compiler translates JML assertions into runtime checks while ESC/Java2 analyzes Java programs at compile time.

To the best of our knowledge no comprehensive assertion package for Go exists that is compatible with the expressive specifications of Gobra. Existing testing tools for Go are not compatible with Gobra because the assertion language typically used by runtime checkers is less expressive than the language used by deductive verifiers. The packages go-contracts [6], gocontracts [21] and godbc [1] provide only pre- and postcondition checks, the package dbc [23] further includes invariants, assertions and assumptions. All checks are purly functional and do not support e.g. predicates, framing or purity assertions as defined in the Section Specification Language. The packages expect the specification to be written as Go code, either by passing the specified properties directly to Go functions that perform the checks [1, 6, 23] or as comments in Go syntax [21]. Hence none of these Go packages can be used as a assertion-based testing supplement to the Gobra verifier.

Core Goals

The Master's thesis results in a Go package that gives users the opportunity for assertion-based testing of their programs. We subdivide the work into the following core goals:

Specification Language

We plan that the specification language used as part of the assertion-based testing framework will be comparable to the specification language of Gobra. Currently, Gobra extends the Go syntax with specification constructs such as invariants or pre- and postconditions. In the future, the Go team wants to

```
1   // require n >= 0
2   // ensure res == n * (n + 1) / 2
3   func sum(n int) (res int) {
4       res = 0
5       i := 0
6       for i <= n {
7           // assert i <= n && res == i * (i - 1) / 2
8           res += i
9           i++
10      }
11      return
12  }
```

Figure 1: Example of functional specifications for a function computing the sum of the first n natural numbers

support the verification of standart Go files, where annotations are written in comments. Our testing framework realizes a first step of this support by introducing the specifications in comments that are going to be used for Gobra.

We do not intend to support the full Gobra syntax. The primary focus is the development of a prototype framework that enables testing *functional* specifications. For an example of functional specifications see Figure 1 that contains an assertion and a pre- and postcondition for a function computing the nth partial sum of the series of natural numbers. This subset of the Gobra syntax can further be improved with support for *purity* assertions. A function is called pure if it does not change the program's state, i.e. if it does not have any side effects. Only pure functions are permitted in assertions. It has to be decided whether to extend the specification syntax with a "pure"-keyword in order to distinguish such functions. We refer the reader to section Runtime Check Generation for a detailed explanation on how to check for purity.

Furthermore, the extraction of functional properties from *predicates* is explored. A predicate is an parameterized assertion. It also needs to be established whether and how we will support exceptional postconditions. In contrast to other programming languages, e.g. Java, recoverable exceptions are implemented using appropriate return values instead of throwing exceptions. Hence, we estimate that no extra syntax is necessary to specify a postcondition in case a recoverable exception occurs. Nevertheless, we will evaluate whether there is a need to specify a postcondition in case a non-recoverable exception (`panic`) occurs. Finally, we plan to include more non-standard specifications for testing frameworks such as labeled old statements and quantifiers. A labeled old statement allows specification with respect to some previous state of the program. Quantifiers reason about elements from a domain, e.g. about all member of a given list. However, unbounded quantifiers will not be considered for runtime checking because one cannot infinitely often instantiate the quantified variable at runtime. Broader Gobra syntax support is later discussed as a possible extension goal, see Section Extended Gobra Syntax Support.

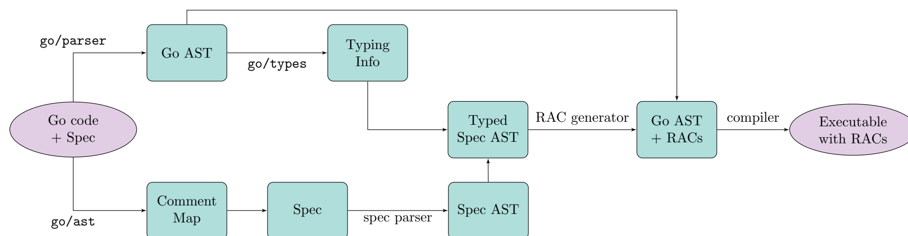

Figure 2: Architecture of the assertion-based testing framework

Assertion-based Testing Framework

An abstract visualization of the assertion-based testing framework's architecture is given in Figure 2. The flow chart shows the conversion of a Go source file including specification annotations into an executable with runtime assertion checks (RACs). Several intermediate steps are performed using existing Go packages:

```go
func sum(n int) (res int) {
    if n < 0 {
        panic("Precondition violated.")
    }
    defer func() {
        if res != n * (n + 1) / 2 {
            panic("Postcondition violated.")
        }
    }()
    res = 0
    i := 0
    for i <= n {
        if i > n || res != i * (i - 1) / 2 {
            panic("Assertion violated.")
        }
        res += i
        i++
    }
    return
}
```

Figure 3: Runtime checks for the specification given in Figure 1

- The Go code is parsed with `go/parser` [13] and `go/ast` [11] in order to receive the Go abstract syntax tree (AST) and a comment map. A comment map holds for every comment the AST node that represents the statement the comment refers to.
- Comments in the comment map that contain specifications are each parsed to a Spec AST.
- The Go AST's typing information is obtained using `go/types` [14].
- The typing information is integrated into the Spec ASTs in order to allow for sanity checks and afterwards to correctly transform the Spec ASTs into RACs.
- Finally, the Go AST is modified to include new nodes that are the translated runtime checks with the help of the `golang.org/x/tools/go/ast/astutil` package [12].
- The result can then be compiled into an executable that will execute the program and perform the runtime checks.

Part of this thesis is spent on combining the existing packages to realize the depicted work flow. Although existing packages can be reused, we estimate that a significant amount of work will be necessary to implement the specification parser and typing the resulting Spec ASTs. The main component of the assertion-based testing package is the conversion of the specification AST into runtime checks which is detailed in the next section called Runtime Check Generation.

Runtime Check Generation

Successfully parsed and typed assertions are integrated as regular Go checks into the Go AST..The final Go AST is compiled into an executable that performs the checks at runtime. Generating Go code for a simple assertion like `assert`

`x != nil` is straight forward: This assertion can be translated to a runtime check as `if (x == nil) { panic("...") }` where the negation of the asserted condition detects any violations of the assertion. Figure 3 pictures a translation of the assertions given in the example in Figure 1. The precondition for `sum` is translated to an if-statement checking the negation of `n` $\geq$ `0` at the beginning of the function. The postcondition is translated to an if-statement whose check is deferred until the end of the function. The assert-statement in the loop is similarly replaced by a runtime check using a simple if-statement.

The design of the conversion from assertions to runtime checks needs to ensure that all runtime checks are side-effect free, i.e. they should not change the behavior of the original program. Additionally, the assertions themselvs need to be side-effect free. This is the reason why only pure functions are permitted in specifications as described in section Specification Language. Checking pureness is non-trivial. We have identified the following approaches:

- The specification language could offer a "pure" keyword allowing programmers to annotate functions. One could either blindly trust such an annotation or check whether the function's implementation is indeed pure.
- The alternative to using pureness annotations is to infer pureness from a function's implementation. Not only the use of impure operations but also determinism has to be checked. Examples for impure statements are memory allocations or I/O operations. Determinism could be checked using a map that stores whether the same input yields the same result at any point in time.

Part of this thesis is to decide which approach to follow. We will also evaluate which approach is more suitable for closing the gap between runtime checking and verification: For instance, it could be confusing for programers to have a function deemed pure for runtime checking but not for verification.

Finally, we plan to support different levels of testing, differentiating the kinds of assertions that are checked at runtime. Examples of such levels can be 1) only assertions, 2) assertions and pre- and postcondtions, 3) every assertion. This feature is motivated by performance considerations since we expect that including runtime checks leads to longer execution times. We will refer to these options as *RAC-generation levels.*

Evaluation

We will evaluate our work based on the following three criteria:

1. Performance overhead of runtime checks
2. The remaining gap between assertion checking and verification
3. Effectiveness of the supported specification language

For the first criterion we measure whether the generation and execution of the runtime checks performs within an acceptable time and memory usage. The perfomance should be within appropriate bounds because otherwise users are demotivated to actually use the tool. Additionally, evaluating the performance overhead of different RAC-generation levels, as described in Section Runtime Check Generation, gives insight into which checks should be performed in a particular level.

```
1          type someStruct struct {
2              val int
3          }
4
5          // require acc(x.val) && n != 0
6          // ensure acc(x.val) && res == x.val / n
7          func (x *someStruct) divideValue(n int) int {
8              return x.val / n
9          }
```

Figure 4: Specification of access permissions

With the second criterion we attempt to decide to which degree we achieved our goal of closing the gap between assertion-based testing and deductive verification. We measure this criterion by annotating Go source code with known bugs first with runtime assertions and then with specification for verification with Gobra. This enables us to evaluate how many additional lines of specification are necessary for verification given that runtime assertions are provided. We can further measure how many lines of specification can be reused from the assertion-based testing.

The third criterion targets the question of how expressive the chosen specification language is. We will need to find a categorization of general classes of specification, and then differentiate which of these are supported by our tool.

Extension Goals

Extended Gobra Syntax Support

The specification language that was described as a core goal, is extended with further Gobra constructs such as checks for *framing* properties. Since we want to support the same specification language as Gobra, the properties we check with assertion-based testing have to be consistent with and expressible in Gobra. Therefore, it needs to be decided how to correctly handle additional constructs: Considering the translation of the access property `acc(x.val)` as seen in Figure 4, we seek to answer questions such as: Is it enough to simply over approximate the assertion by checking whether `x != null` or do we need auxiliary data structures to keep track of currently acquired permissions?

SCION Build System Integration

SCION (scalability, control and isolation on next-generation networks) is a new internet architecture that is designed to be more secure and resilient than today's internet [5]. Parts of SCION are implemented in Go and there already exists an automated testing framework called *scion-fuzz*. It tests SCION's robustness by sending random and potentially malformed inputs to the architectures's interface and thereby discovering crashes or invalid program states [16]. We see the potential of combining scion-fuzz with runtime assertion checks: By translating the specifications, that are present in the implementation, to runtime assertions, scion-fuzz is able to detect more invalid program states. The resulting system, consisting of scion-fuzz and the runtime assertion generator, could be integrated

into the SCION build system to enable continuous testing of the implementation and/against the specification.

Test Input Generation

Manual testing requires programmers to devise inputs for their tests. This effort can be reduced by automatically generating test input. We can also give better guarantees on program correctness if the runtime checks generated from assertions succeed provided different inputs. A technique for generating test input has been mentioned in the extension goal of the SCION Build System Integration: *Fuzzing*. This is available for Go as the package `gofuzz` [22]. Another option would be to use *Property-based Testing*, a testing technique that tries to falsify a given property by generating random input data and verifying the expected behaviour [7]. An extension goal for this thesis is the assessment of different test input generation strategies. Subsequently, we will incorporate the best performing input generation into our testing framework. This permits an additional evaluation of the assertion-based testing framework: Using the testing framework and input generation, we can count the number of errors found with assertion-based testing of annotated Go source code with known bugs.

Negative Testing

Similar to code, specification can also contain bugs. In particular, specifications might not capture the behavior a user intended to specify. Therefore, it is important to test the correctness of the specification before attempting a verification. A technique that has been used for specification testing is *Negative Testing*. As described in [10], Negative Testing actively introduces bugs into the implementation to check whether the specification can detect these bugs during verification. The same approach can be followed by our runtime assertion framework. Hence, this extension goal is to combine the assertion-based testing framework with a Negative Testing strategy.

References

[1] Junade Ali. Go by contract. www.github.com/IcyApril/gobycontract. [Online; accessed 2020-05-08].

[2] Linard Arquint, Martin Clochard, Peter Müller, Wytse Oortwijn, and Felix Wolf. Gobra. www.pm.inf.ethz.ch/research/gobra.html. [Online; accessed 2020-05-14].

[3] John Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley Longman Publishing Co., Inc., USA, 2003.

[4] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[5] David Barrera, Laurent Chuat, Adrian Perrig, Raphael Reischuk, and Pawel Szalachowski. The scion internet architecture. *Communications of the ACM*, 60:56–65, 05 2017.

[6] Sergey Bronnikov. go-contracts. www.github.com/ligurio/go-contracts. [Online; accessed 2020-05-08].

[7] Koen Claessen and John Hughes. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP*, 46, 01 2000.

[8] Lori Clarke and David Rosenblum. A historical perspective on runtime assertion checking in software development. *ACM SIGSOFT Software Engineering Notes*, 31:25–37, 05 2006.

[9] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. *SIGPLAN Not.*, 37(5):234–245, May 2002.

[10] Pedro Fonseca, Kaiyuan Zhang, Xi Wang, and Arvind Krishnamurthy. An empirical study on the correctness of formally verified distributed systems. pages 328–343, 04 2017.

[11] The Go Authors (golang.org/AUTHORS). Go package ast. www.golang.org/pkg/go/ast. [Online; accessed 2020-05-18].

[12] The Go Authors (golang.org/AUTHORS). Go package astutil. www.pkg.go.dev/golang.org/x/tools/go/ast/astutil. [Online; accessed 2020-05-14].

[13] The Go Authors (golang.org/AUTHORS). Go package parser. www.golang.org/pkg/go/parser. [Online; accessed 2020-05-14].

[14] The Go Authors (golang.org/AUTHORS). Go package types. www.golang.org/pkg/go/types. [Online; accessed 2020-05-14].

[15] The Go Authors (golang.org/AUTHORS). Go: The programing language. www.golang.org/. [Online; accessed 2020-05-07].

[16] Alexander Kunze. *Efficient Automated Testing Framework for the SCION Architecture*. Bachelor Thesis, Network Security Group, Department of Computer Science. Eidgenoessische Technische Hochschule Zuerich, 2019.

[17] Gary Leavens, Albert Baker, and Clyde Ruby. Jml: A notation for detailed design. 1970.

[18] Gary Leavens and Yoonsik Cheon. Design by contract with jml. 2004.

[19] Bertrand Meyer. *Object-oriented software construction*. Series in Computer Science. Prentice-Hall International, 1988.

[20] P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In M. Butler and W. Schulte, editors, *Formal Methods (FM)*, volume 6664 of *Lecture Notes in Computer Science*, pages 73–87, 2011.

[21] Marco Ristin. gocontracts. www.godoc.org/github.com/Parquery/gocontracts/gocontracts. [Online; accessed 2020-05-08].

[22] Daniel Smith and Tim Hockin (Google). gofuzz. www.github.com/google/gofuzz. [Online; accessed 2020-05-18].

[23] Ruslan Stepanenko. Design-by-contract for go1. www.godoc.org/github.com/drblez/dbc. [Online; accessed 2020-05-08].