AALBORG UNIVERSITY
STUDENT REPORT

P10

Group cs-24-ds-10-03

Department of Computer Science

Aalborg University

AALBORG UNIVERSITY
STUDENT REPORT

Department of Computer Science
Aalborg University
Selma Lagerløfs vej 300
https://www.cs.aau.dk

Title:
DropShadow: Instrumenting Go with Contracts and Automatic Property-Based Test Generation of Hyperproperties

Project Period:
Spring Semester 2024

Project Group:
cs-23-ds-10-03

Participant(s):
Andreas Kjeldgaard Brandhøj
Dat Tommy Thanh Dieu
Kasper Vesteraa

Supervisor(s):
René Rydhof Hansen
Danny Bøgsted Poulsen

Page Numbers:
60

Date of Completion:
May 22, 2024

Abstract:

This report presents an exploration and evaluation of DropShadow: a testing framework, leveraging contract-driven testing which automatically generates and executes property-based tests for the Go programming language. The tool integrates contracts written by the developer directly into Go source code and allows for the automatic generation of tests, checking both trace- and hyperproperties; the latter achieved through sequential self-composition. This integration aims to facilitate property-based testing, reducing the required manual work for writing tests, as well as, introduce more comprehensive automated checks for program correctness against specified behaviours. Moreover, this contractual integration ensures not only individual functions comply with their specifications, but also functions calling or called by said functions, hence offering a cascading assurance of reliability throughout the program. Our evaluation shows a potential practical application of the tool in different settings through the use of various constructs with the added benefit of mitigating contractual explosion.

Summary

This project is based on experiences from our P9 project *Fuzzy Security*, which explored the application of fuzzing techniques for enhancing cyber resilience and software robustness in general. In this context, fuzz testing on Go code provided by the company DEIF was conducted. From these experiences, investigations into property-based testing for the Go programming language started, leading us to explore *trace properties*, *hyperproperties* [6], and *contractual design* as well as how these concepts can be used to reason about the correctness of a program. Advancing into the domain of hyperproperties allows us to utilise, relative to trace properties, more expressive powers, as it includes quantifiers over sets of traces. This is essential for expressing some important properties such as non-interference.

One way to specify properties is through contracts, as presented in the Eiffel programming language, which introduced the concept of *Design-By-Contract* [7]. However, during our research, we found the only available tools for specifying contracts in Go to be *gocontracts* [28] and *dbc4go* [3]. These tools allow the developer to express properties in the form of pre- and post-conditions in contracts, written as comments above functions. These comments are then parsed and instrumented in the function body as run-time assertion checks. However, we believe the use of contracts can be further expanded with hyperproperties.

We propose a tool called DropShadow, which extends the use of contracts in the Go programming language with hyperproperties. We introduce automatic test case generation derived from contracts by annotating functions with comments specifying properties. Using a DropShadow syntax, we enable developers to express both trace- and some reducible hyperproperties, which are then tested. Instead of specifying properties directly in the test cases (following the typical *Arrange, Act, Assert* structure), properties are expressed in the contracts and injected directly into the function body. This approach differs from existing property-based testing tools in Go, where the properties are specified in the test cases themselves. DropShadow in short comprises of: 1) parsing of annotated functions 2) extraction of annotations into contracts 3) injection of these contracts into source code 4) generation of tests based on these contracts and 5) evaluation of these tests.

The main contribution of this project is the DropShadow tool, which is presumably the first tool designed to integrate contractual design alongside automatic test generation tailored to the Go programming language. Furthermore, DropShadow incorporates contracts encompassing hyperproperties, thus enhancing its capability to express security properties over multiple traces.

Contents

Chapter 1

Introduction

In the field of software engineering, there is a continuous search for methodologies to enhance the robustness and reliability of software systems. Amidst these endeavours, hyperproperties emerge as a crucial concept, particularly useful in defining some security policies, since they provide a formalisation for specifying and reasoning about the relationships between multiple execution traces of a system [6]. With it, properties such as non-interference, ensuring high-security operations do not influence or interfere with low-security operations, can be expressed. This principle is not merely theoretical but is widely applied in practical scenarios, such as in critical infrastructure, financial, and healthcare systems, where security and data integrity are paramount.

An example of this is The Heartbleed bug [34], a bug where low inputs, in the form of heartbeat requests, should only result in low outputs. However, a problem arose due to missing input validation, leading to the leakage of private sensitive information being transmitted to the requester, which could be a private key [26]. This case highlights a real occurrence of interference bugs; however, due to its nature, a single trace could lead to its discovery by using address sanitisers in conjunction with, e.g., OSS-Fuzz [27]. Be that as it may, not all leaks are caused by reading memory outside bounds. Incorrect branching can also result in a low output being assigned a high value.

The motivation behind this project is partially rooted in some of the limitations of traditional unit testing and property-based testing frameworks encountered, as well as the formalisation of functional specifications in software development. Unit testing, while effective for testing individual units of code under some predetermined conditions, can fail to capture complex interaction patterns or unexpected input combinations occurring in real-world scenarios. It might be the case, that a unit-tested function interacts with other components in unspecified ways, but still yields a satisfactory result. Instead, functional specifications in the form of contracts written as documentation can capture incorrect usages and be used to generate tests. The problems are not least contingent on concepts like the *pesticide paradox* (diminishing effectiveness of test cases over time) as well as *bias and blind spots* (the tendency for developers to write tests, that conform to the expected behaviour of the code) [18]. Property-based testing might accommodate some of these limitations by utilising an approach for checking properties across a broad range of inputs (often through the generation of (pseudo) random inputs).

Contractual design of functions often only concerns the relation between inputs and outputs. Many tools already exist for contractual specification of functions in their documentation as well as for property-based testing, some of which are gocontracts, QuickCheck [4], Hypothesis [23], among others. To our knowledge, however, no tool for contractual design supporting some hyperproperties over the universal quantification of two pairs of traces with automatic test case generation exists - not least for the Go language. Thereby, manual effort is required to extend the function to call itself twice to perform hypertesting [21].

Our main contribution is the tool DropShadow, which implements contractual specification of functions with automatic test generation of both trace- and hyperproperties. To support it, a wide range of input generators are provided with a general interface enabling developers to extend its capabilities. We solve some general problems in which other tools are hindered, such as multiple return statements and how hyperproperties can be defined as contractual obligations. To support our work, we present how assertions, trace properties, and hyperproperties interrelate, and how sequential self-composition can be applied in some circumstances. We also present general patterns we have identified concerning the use of contractual design.

Chapter 2

Theory

This chapter seeks to lay the foundation for understanding some of the essential concepts and theories that underpin our project. It begins with an exploration of programming with assertions, a technique that integrates explicit statements about program states to ensure correctness. We then delve into trace properties and hyperproperties, which encompass properties of sets of computations traces, offering a broader perspective on system behaviours. Finally, we examine sequential self-composition, which is an approach to reduce a subset of hyperproperties to trace properties.

2.1 Programming With Assertions

In this section, we focus on how it is possible to articulate important program properties. For many programming languages, they can be expressed through the use of *assertions* [25]. An assertion is often a Boolean-valued function over the state space, which is usually expressed as a logical proposition using the program's variables. Such a proposition could look like this:

$$x + y > 3$$

which may or may not be satisfied by a given state of the program during execution. For instance, if both x and y equal 4, the assertion is satisfied, whereas this is not the case if both variables equal 1.

An assertion failure typically implies an undesired program state, in which case an immediate termination of the program (often accompanied by an error message indicating the location of the failure) can be triggered. These precautions can help identify bugs and other inexpediencies by highlighting, where the program's actual state deviates from the expected one defined by the assertion.

2.1.1 Pre- and Post-conditions

However, assertions alone are not sufficient to arrive at a satisfactory expression of program correctness. While they do serve to confirm individual properties at particular points, they do not address the entirety of a program's execution or its interactions with external components. We need to look at the properties of a program fragment (i.e. a function) with respect to the assertions, that are satisfied before and after execution of that fragment [25]. Therefore, two kinds of assertions must come into play:

- Pre-conditions: propositions assumed to be satisfied by the *caller* before the given fragment of code is executed.
- Post-conditions: propositions expected to be satisfied by the *callee* after the given fragment of code is executed.

In other words, a program fragment *C* is said to be correct with respect to a certain pre-condition *P*, and a certain post-condition *Q*, if and only if for all states satisfying *P* a state is produced satisfying *Q*. This can be formally expressed as the Hoare triple [25]:

$$\{P\}C\{Q\} \tag{2.1}$$

In other words - it is the responsibility of the environment to ensure, that a program fragment is only executed when that fragment's pre-condition is met. In turn, the fragment is obligated to fulfil the post-condition.

To put it in context: specifying pre- and post-conditions is like a *contract* between the environment (caller) and the program fragment (callee). The pre-condition obligates the environment, and the post-condition obligates the program [25]. If the environment does not fulfil its part of the deal, the program may do what it likes. On the other hand, if the pre-condition is satisfied and the program fails to ensure the post-condition, the program is deemed incorrect.

It is important to notice, that program correctness is only a relative concept according to Meyer [25]; there is no such thing as an intrinsically correct or incorrect program. It only makes sense to talk about the correctness of a program with respect to a certain specification given by a set of pre- and post-conditions. Or, as Malloy and Voas state: "*The program is correct if its implementation is consistent with the assertions.*" [24]

2.1.2 Example

Assertions in programming serve as explicit checkpoints, allowing developers to enforce expected conditions or invariants. They are thus powerful tools for improving software quality by explicitly stating conditions, that must hold true at specific points in the execution. Assertions can not only help a programmer read the code and understand the valid/invalid flow of information through it, but they can also help the program detect its defects. When an assertion fails, it signals a discrepancy between the expected and actual program state, effectively identifying a potential defect within the program's logic or state. This characteristic led Rosenblum to call programs with assertions "*self-checking programs*" [32].

```go
1  func BMO(low, high int) int {
2      if high < 0 {
3          panic("pre-condition violated")
4      }
5      intermediary := 0
6      if high % 2 == 0 {
7          intermediary = 271
8      }
9      result := low
10     result += intermediary
11     if result < 0 {
12         panic("post-condition violated")
13     }
14     return result
15 }
```

Listing 2.1: Example of using assertions in the code.

Listing 2.1 shows an example of using panics in Go. Go does not directly support assertions, however - panics can be used to communicate an exceptional violation (like exceptions in other languages). The `BMO` function receives two parameters `low` and `high` and returns one value. In this example, we see how assertions are used to check the pre- and post-conditions of the function, hence ensuring the validity of the information flowing into as well as out of `BMO`.

In line `2` the pre-condition is checked to ensure the input variable `high` is positive as expected by the function. If this is not the case, a panic is raised, informing about the failed pre-condition. Likewise, in line `11` we see an assertion checking the post-condition of, which in this case ensures, that the result is positive. Again, if this does not hold true, a panic is raised informing about the failed post-condition.

2.2 Trace Properties

As described in the previous section, assertions are used to describe propositions (in the language of the program), thus denoting the validity of the program's state. This works on a snapshot - a state captured at the location of the assertion. However, these assertions do not consider the previous ones or how the execution got to them. To accommodate this problem, trace properties will be presented following the notations in [6].

An atomic proposition is a basic, indivisible statement in logic, that expresses a fact or a condition without any further decomposition. It is deemed *atomic*, since it cannot be broken down into simpler statements and is either true or false, without any ambiguity or need for further interpretation.

To define programs and how we describe states and their corresponding atomic propositions, we use a Kripke structure, which is a transition system with a labelling function. Let a system, or program, be modelled as a non-empty set of infinite traces from a Kripke structure $K = (S, s_0, \delta, \mathbf{AP}, L)$. S is the set of states, s_0 is the initial state, $\delta : S \to \mathcal{P}(S)$ the transition function which for any state has a transition, **AP** the atomic propositions, and $L : S \to \mathcal{P}(\mathbf{AP})$ labelling a state with its corresponding atomic propositions. A path in this structure is an infinite sequence of states [5].

With this, let the set of all atomic propositions in a state be expressed as $\Sigma = \mathcal{P}(\mathbf{AP})$. In relation to actual programs, the finite traces, which are the executions that terminate, are defined as Ψ_{fin}. The infinite traces Ψ_{inf} are the executions, which do not terminate.

$$\begin{aligned}\Psi_{\mathit{fin}} &\triangleq \Sigma^* \\ \Psi_{\mathit{inf}} &\triangleq \Sigma^\omega \\ \Psi &\triangleq \Psi_{\mathit{fin}} \cup \Psi_{\mathit{inf}}\end{aligned}$$

where Σ^* is the Kleene star operation performed on Σ, creating the set of all traces with a finite length, and Σ^ω is the traces with infinite length. All traces Ψ are then the union of both the finite and infinite traces, where the finite traces are made infinite by stuttering, or repeating, the final state of the execution of the program. This is necessary as the Kripke structure contains only infinite and possible traces, since its transition function always has a transition, which we, for stuttering, require to not transition to another state. With this, we can define a trace t as an infinite sequence of states $t = s_0 s_1 ...$ where $s_i \in \Sigma, i \in \mathbb{N}_0$. However, for brevity, we consider traces as a sequence of atomic proposition as if the labelling function has been applied to all states; with the exception, of when we strictly mention that traces are a sequence of states. Relevant later, in the context of sequential self-composition, is the concatenation of traces, which is done with a finite trace t and potentially infinite trace t' is denoted as tt'.

$$\begin{aligned}t[i] &\triangleq s_i \\ t[: i] &\triangleq s_0 s_1 \cdots s_i \\ t[i :] &\triangleq s_i s_{i+1} \cdots\end{aligned}$$

Using traces we can now define trace properties: a proposition over a set of traces allowing for the categorisation of valid and invalid traces. Like state assertions, which can allow multiple concrete states to be acceptable, a trace property can also allow multiple traces to be acceptable. The set of traces T that satisfy the trace property P is denoted as $T \models P$ if and only if all the traces in T are in P. Another way of looking at it is stating, that T is all the traces of the implementation, and P is those of the specification. The implementation only implements P if and only if all traces in the implementation are in the specification.

$$T \models P \triangleq T \subseteq P$$

The trace property, or a set of traces satisfying a property, is part of a universe. This universe $\mathbf{P}$ is the set of all possible infinite traces, which is the powerset of all infinite traces. The only trace property that fully encompasses the universe is one that is satisfied for all infinite traces.

$$\mathbf{P} \triangleq \mathcal{P}(\Psi_{inf})$$

With traces and trace properties defined, we will now present a way of expressing trace properties with the use of Linear Temporal Logic.

2.2.1 Linear Temporal Logic

One way of formally writing trace properties is using Linear Temporal Logic (LTL) [31]. ”Linear” refers to reasoning over a sequential order of the propositions in the trace and ”Temporal” to a progression in discrete time, that allows the expression of different points or indices of a trace. LTL enables expressing trace properties for non-branching systems. That is systems where the future behaviour is solely determined by the current state. Below, a version of the grammar and semantics is presented, where ψ denotes the production rule of LTL and ap an atomic proposition [17, 9].

$$\psi := ap|\neg\psi|\psi_1 \wedge \psi_2|X\psi|\psi_1 \cup \psi_2 \tag{2.2}$$

$$\begin{aligned}
&t \models ap && \text{if } ap \in t[0] && (2.3)\\
&t \models \neg\psi && \text{if } t \not\models \psi && (2.4)\\
&t \models \psi_1 \wedge \psi_2 && \text{if } t \models \psi_1 \text{ and } t \models \psi_2 && (2.5)\\
&t \models X\psi && \text{if } t[1:] \models \psi && (2.6)\\
&t \models \psi_1 U \psi_2 && \text{if } \exists i \geq 0.\ t[i:] \models \psi_2 \text{ and } \forall 0 \leq j < i.\ t[j] \models \psi_1 && (2.7)
\end{aligned}$$

For example, Equation 2.6 means, that the trace t must satisfy the trace property ψ at the ne*X*t index $t[1:]$ immediately following the current one, and Equation 2.7 means that the trace t to satisfy $\psi_1 U \psi_2$, ψ_1 must hold *U*ntil ψ_2 holds.

An example of these rules can be seen in Figure 2.1 and Figure 2.2. In this and future examples, we have taken the liberty to write $\{\psi\}$ as the set of atomic propositions for ψ to hold at a given time within a trace. This disregards, that ψ is not necessarily an atomic proposition; it is done for conciseness and as a simplification. Alternatively, $\{\neg\psi\}$ is the set of atomic propositions for ψ not to hold.

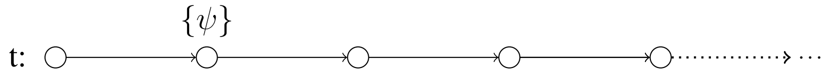


Figure 2.1: An example of $t \models X\psi$ where ψ is satisfied at the ne*X*t index in the trace.

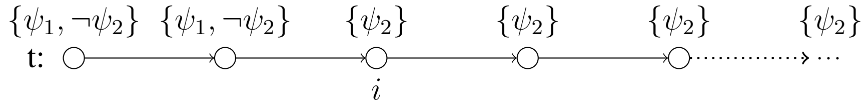


Figure 2.2: An example of $t \models \psi_1 U \psi_2$ where ψ_1 is satisifed *U*ntil i from which ψ_2 is satisfied indefinitely.

In addition to the previously defined semantic rules, we can deduce other operators [9].

$$\psi_1 \vee \psi_2 \equiv \neg(\neg\psi_1 \wedge \neg\psi_2) \tag{2.8}$$

$$\psi_1 \implies \psi_2 \equiv \neg\psi_1 \vee \psi_2 \tag{2.9}$$

$$\psi_1 \iff \psi_2 \equiv \psi_1 \implies \psi_2 \wedge \psi_2 \implies \psi_1 \tag{2.10}$$

$$true \equiv ap \vee \neg ap \tag{2.11}$$

$$false \equiv \neg true \tag{2.12}$$

$$\psi_1 R \psi_2 \equiv \neg(\neg\psi_1 U \neg\psi_2) \tag{2.13}$$

$$F\psi \equiv true U \psi \tag{2.14}$$

$$G\psi \equiv \neg F \neg \psi \tag{2.15}$$

An example of Equation 2.13, Equation 2.14, and Equation 2.15 can be seen in Figure 2.3, Figure 2.4, and Figure 2.5 respectively.

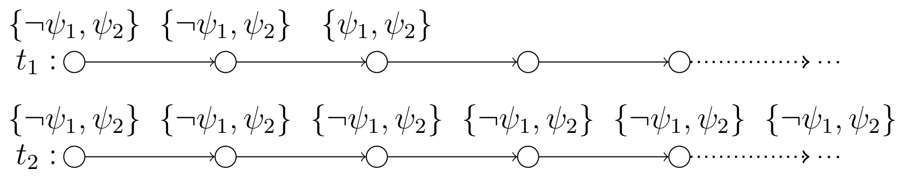


Figure 2.3: An example of $t_1 \models \psi_1 R \psi_2$ where ψ_2 is satisfied until it is *R*eleased from which point ψ_1 is satisfied; and $t_2 \models \psi_1 R \psi_2$ where ψ_2 is satisfied indefinitely.

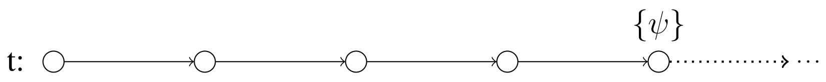


Figure 2.4: An example of $t \models F\psi$ where ψ is eventually (*F*inally) satisfied.

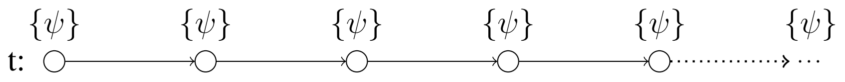


Figure 2.5: An example of $t \models G\psi$ where ψ is always (*G*lobally) satisfied.

Next, we will apply Linear Temporal Logic (LTL) to describe a trace property of the function from Listing 2.1.

2.2.2 Example

Consider the function from Listing 2.1: now we want to use this example to show, how trace properties can be used to describe expected behaviour based on the inputs low and $high$ and output o. For a function to be callable, its pre-conditions must be met, in which case eventually the post-conditions should be satisfied after the function's execution. This can be described as follows:

- *Region 1* (High is even): if the input `high` is greater than or equal to 0 and it is even, then the output is $\texttt{low} + 271$.
- *Region 2* (High is odd): if the input `high` is greater than or equal to 0 and it is odd, then the output is $\texttt{low} + 0$.
- Provided the input `high` is always greater than or equal to 0, then the output will never be negative. Therefore, the post-condition will never be violated.

The two first cases describe valid executions of the function from Listing 2.1. For each execution, we have given a shorthand: *region 1* and *region 2*. We use these to encapsulate how possible overlapping input sets describe a mapping to some output. This approach is similar to partition testing; we can thus separate each region into its pre- and post-condition:

$$P_1 = high \geq 0 \wedge high \bmod 2 = 0 \tag{2.16}$$
$$Q_1 = output = low + 271 \tag{2.17}$$
$$P_2 = high \geq 0 \wedge high \bmod 2 = 1 \tag{2.18}$$
$$Q_2 = output = low + 0 \tag{2.19}$$

Region 1 is described by the pre-condition Equation 2.16 and post-condition Equation 2.17. *Region 2* is described by the pre-condition Equation 2.18 and post-condition Equation 2.19. The pre-conditions are not atomic propositions but are instead composed of two atomic propositions.

We can see the trace property for *Region 1* in Equation 2.20 and *Region 2* in Equation 2.21. They both describe how satisfying their pre-condition implies, that the post-condition is eventually satisfied.

$$reg_1 = P_1 \implies F(Q_1) \tag{2.20}$$
$$reg_2 = P_2 \implies F(Q_2) \tag{2.21}$$

Equation 2.22 describes the trace property for the function `BMO` in Listing 2.1. It describes a trace property consisting of all traces satisfying at least one region's pre-condition. For this satisfied pre-condition, the corresponding region's post-condition must be eventually satisfied.

$$\psi = (P_1 \vee P_2) \wedge (reg_1 \wedge reg_2) \tag{2.22}$$

Without $(P_1 \vee P_2)$, inputs not specified by at least one region would be in the trace.

2.3 Hyperproperties

In [6] trace properties are extended to hyperproperties, where each hyperproperty is a set of sets of infinite traces or trace properties. A hyperproperty can then define a relation between multiple trace properties. This is necessary to express e.g. non-interference and observational determinism, which can not be checked on a single trace, as it requires the comparison of different executions.

All hyperproperties **HP** is the powerset of all trace properties **P**, which is the same as the powerset of powerset of all infinite traces ψ_{inf}. The definition is as follows:

$$\mathbf{HP} \triangleq \mathcal{P}(\mathcal{P}(\Psi_{\text{inf}})) = \mathcal{P}(\mathbf{P}). \tag{2.23}$$

Each element in the set **HP** is a hyperproperty H, which is a set of trace properties **P**. Thereby, every trace property in a hyperproperty is an allowed system, that specifies which executions must be possible.

$$T \models H \triangleq T \in H \tag{2.24}$$

That is, the set T of traces satisfies hyperproperty H, denoted as $T \models H$, if and only if, the trace property T is in H. It looks similar to trace properties, however, the main difference is, that trace properties require T to be $\subseteq$ of P while hyperproperties require T to be $\in H$.

2.3.1 HyperLTL

After having defined hyperproperties, a closer look will now be taken on Hyperproperties Linear Temporal Logic (HyperLTL), which is a language used to formulate hyperproperties. Unlike LTL, which expresses properties of single execution traces over time, HyperLTL allows the specification of properties across multiple traces by introducing quantifiers. This capability makes it well-suited for expressing properties such as non-interference, as well as other properties involving correlations between different execution paths or system behaviours [6].

In the context of HyperLTL, quantifiers allow for the specification of properties, that must hold for all possible combinations of traces or some combination of these, enabling the expression of complex *inter-trace* properties. The syntax of HyperLTL can be defined by the following grammar [5]:

$$\Upsilon ::= \exists \pi . \Upsilon \mid \forall \pi . \Upsilon \mid \psi \tag{2.25}$$

$$\psi := ap^{\pi} \mid \neg \psi \mid \psi_1 \wedge \psi_2 \mid X\psi \mid \psi_1 U \psi_2 \tag{2.26}$$

where $\pi \in \mathcal{V}$ is a trace variable, and $\mathcal{V}$ is an infinite supply of trace variables π_1, π_2, π_3, ... $\exists$ (existential quantifier) applied on π, suggests that there is at least one trace for which the predicate holds true, while $\forall$ (universal quantifier) indicates that the predicate following is true for all traces. Equation 2.25 thus shows, that HyperLTL extends LTL by introducing quantification over traces, enabling the expression of properties across multiple traces, allowing the expressivity required for some information-flow properties like non-interference. In addition, Equation 2.26 is extended such that the atomic proposition ap refers to a trace π.

The validity of HyperLTL formulas is written $\Pi \models_T \Upsilon$, where T is the set of traces. The trace assignment as a partial function maps trace variables to traces $\Pi : \mathcal{V} \to \Psi$, where Ψ is the set of all traces and, $\Pi[\pi \to t]$ denotes that π is mapped to t:

$$\Pi \models_T \exists\pi.\Upsilon \quad \text{if } \exists t \in T : \Pi[\pi \mapsto t] \models_T \Upsilon \tag{2.27}$$

$$\Pi \models_T \forall\pi.\Upsilon \quad \text{if } \forall t \in T : \Pi[\pi \mapsto t] \models_T \Upsilon \tag{2.28}$$

$$\Pi \models_T ap^\pi \quad \text{if } ap \in \Pi(\pi)[0] \tag{2.29}$$

$$\Pi \models_T \psi_1 \wedge \psi_2 \quad \text{if } \Pi \models_T \psi_1 \text{ and } \Pi \models_T \psi_2 \tag{2.30}$$

$$\Pi \models_T X\psi \quad \text{if } \Pi[1:] \models_T \psi \tag{2.31}$$

$$\Pi \models_T \psi_1 U \psi_2 \quad \text{if } \exists i \geq 0 \text{ s.t. } \Pi[i:] \models_T \psi_2 \text{ and } \forall j < i, \Pi[j:] \models_T \psi_1 \tag{2.32}$$

For instance, the second rule (Equation 2.28) describes the semantics of universal quantification over traces, and can be read as follows: $\Pi \models_T \forall\pi.\Upsilon$ means that for the trace assignment Π and a set of traces T the formula $\forall\pi.\Upsilon$ is satisfied if for every trace t in T, the formula Υ is satisfied when a unique π is mapped to t in Π. Put in simpler terms, this rule ensures, that the property described by Υ must hold for every individual trace in the set T, thereby defining a property of the system, that applies universally across all possible executions of behaviours captured by T. $\Pi[n:]$ denotes the trace assignment Π where each π maps Π to the trace $\Pi(\pi)[n:]$. [5]

Similarly, the fourth rule (Equation 2.30) expresses the semantics of the logical conjunction (AND) operator within the context of HyperLTL, applied to the formulas ψ_1 and ψ_2. It can be read as follows: for a set of traces T and a given trace assignment Π, the formula that ψ_1 and ψ_2 are true is satisfied if it is the case, that ψ_1 is satisfied, and ψ_2 is satisfied under that trace assignment.

In [5], syntactic sugars for comparing traces have been defined. With a set of atomic propositions AP, $\pi[0] =_{AP} \pi'[0] \equiv \bigwedge_{ap \in AP} ap^\pi \leftrightarrow ap^{\pi'}$, meaning comparing a common set of atomic propositions AP between two traces (π and π') in the initial state is done by checking whether the atomic propositions in AP are all equal. In addition, we have $\pi =_{AP} \pi' \equiv G(\pi[0] =_{AP} \pi'[0])$.

In Section 2.2 we covered trace properties, wherein additional operators, such as *G*lobally (G), were defined with logical equivalence. The previous equations for hyperproperties' semantics cover the same fundamental operators as trace properties. For this reason, we apply the same logical equivalences on hyperproperties to define the same operators, given that an atomic proposition ap now applies to π as shown in Equation 2.26.

With the syntax and semantics for HyperLTL defined, we can now formulate the previously mentioned hyperproperties (non-interference and observational determinism) in this temporal logic.

Non-interference is a property which states, that for any computation traces, if all inputs, except for high-security inputs, are identical across the traces at all time, the outputs must also remain indistinguishable at all times [10]. Given a concrete execution of a program, inputs and outputs have concrete valuations, then we can construct two sets of atomic propositions describing their equality. I_L is the atomic propositions over the low inputs, and O the atomic propositions over the outputs. This can be expressed as:

$$\forall\pi.\forall\pi'.\pi =_{I_L} \pi' \Rightarrow \pi =_O \pi' \tag{2.33}$$

where π and π' represent two arbitrary computation traces, and *H* is the high-security inputs whose influence on the low-security output *o* is being examined. The universal quantifiers $\forall\pi$ and $\forall\pi'$ signify, that the statement must hold for every possible pair of traces.

Observational Determinism: Expressing observational determinism in HyperLTL involves stating, that for every pair of traces observed under identical conditions, the outcomes must be the same [9]. An example of observational determinism can be found in the field of software testing, where a program should consistently produce the same outcomes when the same test conditions are applied. Given *I* is the atomic propositions over all inputs. This can be capture with the following HyperLTL formula [9]:

$$\forall\pi.\forall\pi'.\pi =_O \pi' \Rightarrow \pi =_I \pi' \tag{2.34}$$

This formula ensures, if two traces produce the same outputs throughout their execution, then their inputs must have been identical at all points in time.

2.3.2 Example

In Section 2.2.2 we argued how the trace property of the function `BMO` from Section 2.1.2 could be defined. Now, we would like to see if there is interference between the low output and the high input, as well as observational determinism. For that, we cannot use the aforementioned trace property.

Non-interference: we define the set of inputs $I = \{low, high\}$ and high inputs $H = \{high\}$. Then we must search for two executions with varying high inputs and different outputs. If that is the case, then we have interference where information flows from a high input to a low output. By looking at the function in Listing 2.1 we can see that the value of `intermediary` is dependent on the value of the input `high`. We can then prove interference by these two executions `BMO(0, 0)` with output `271` and `BMO(0, 1)` with output `0`.

Observational determinism: with the same inputs *I* we can observe non-deterministic behaviour by looking at a function's input and outputs across two executions. By manual inspection of the code for `BMO`, we see that its definition is inherently deterministic, and therefore we cannot find a counter-example for observational determinism.

The function `BMO` is observationally deterministic but contrary to its name, it does exbibit interference. We showed testing for interference can be done by executing `BMO` twice and comparing the outputs. In the following section, we will go into more detail about how and why this works.

2.4 Sequential Self-Composition

HyperLTL is a general language used to express hyperproperties. By focusing on functions with terminating traces, we can consider a subset of hyperproperties, described by finite executions. By using trace concatenation, we can then combine the two finite traces into one trace. This allows us to consider one finite trace, in contrast to two individual finite traces, consequently reducing a hyperproperty to a trace property.

The common *Arrange-Act-Assert* paradigm assumes a pre-condition is satisfied after executing the *Arrange* step, some functionality is performed in *Act*, and the expected post-condition is asserted in the *Assert* step. This process involves a single execution of *Act* and essentially functions as a trace property. By extending this approach, we can append another execution of *Act*, which depends on yet a new *Arrange* step. Consequently, this necessitates a new *Assert* step, which now considers the input-output relation from both *Act* steps. This results in a new *Arrange-Act-Arrange'-Act'-Assert'* structure.

Formally, this approach is a sequential self-composition of a program P, where P' denotes P with all variables renamed (primed), to ensure uniqueness. $P; P'$ is the sequential execution of P followed by P'. As an example, let P be a declaration and an assignment, where the value returned from the function `BMO` is assigned to `o`, such that `o := BMO(l, h)`. Correspondingly, P' would be `o' := BMO(l', h')`. The composition results in a program that first executes `BMO(l, h)` followed by `BMO(l', h')`. Later this will be presented in regard to Listing 2.1 [6]

However, the variables must be uniquely declared and assigned appropriately to the specified relation. Therefore, we consider the program P as both the Arrange-Act parts and P' as Arrange'-Act'. Important for hyperproperties is the ability to describe how inputs relate to each other. The primed inputs might be identical to, modified versions of, completely different from, or partially related to the original inputs. The *Arrange'* is responsible for defining this relationship.

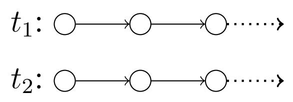
Consider again the example `o := BMO(l, h)` and `o' := BMO(l', h')`. Suppose we want the inputs to be related as follows: `l = random`, `h = random`, `l' = l` and `h' = random`. The assertion would then be `o = o'`. Written as a test case, this would follow an *Arrange-Act-Arrange'-Act'-Assert'* structure, seen in Listing 2.2.

```
l, h := random, random
o := NI(l, h)
l', h' := l, ranodm
o' := NI(l', h')
assert o = o'
```

Listing 2.2: A test case where sequential self-composition is used to reduce non-interference to a trace eproperty. It follows the structure of Arrange-Act-Arrange'-Act'-Assert'.

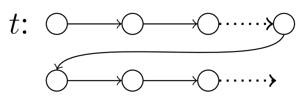
Figure 2.6 shows a representation of sequential self-composition by concatenating two traces together.

```
out := NI(l, h)
```



(a) Two separate executions, or traces t_1 and t_2, of NI.

```
out := NI(l, h)
out' := NI(l', h')
```



(b) Two separate executions of `NI` with sequential self-composition concatenating the two traces together to one.

Figure 2.6: Non-interference reduced to a trace property using sequential self-composition of the function `NI` (See Listing 2.1) where the inputs and outputs are renamed to unique identifiers by priming them.

By concatenating two traces into one, we can utilise trace properties to check certain hyperproperties. However, some hyperproperties may require more than two traces. In such cases, we follow the same procedure, appending additional primes to create unique identifiers and executing the function a third, fourth or more times. For sequential self-composition to work, several practical assumptions are necessary: the executions of the sequential functions must operate on the same initial state, all concurrent executions that could interfere with future self-composed executions must be completed, and primed variables should guarantee uniqueness.

Earlier, assertions were described as a language construct used to express function contracts. These contracts can formalise pre-and post-conditions, which fundamentally are trace properties. However, not all properties can be expressed with one trace (non-interference requires at least two, for example). In practice, testing hyperproperties is challenging, but sequential self-composition can reduce some hyperproperties to trace properties. This allows us to use trace properties to express e.g. non-interference. In the following chapter, the tool DropShadow will be presented by applying the theory in practice.

Chapter 3

DropShadow

In this chapter, we are going to present our tool DropShadow[1] written in Go. Figure 3.1 depicts the overall workflow. We will first summarise the main components and explain the steps involved; then, in the following sections, the details will be covered.

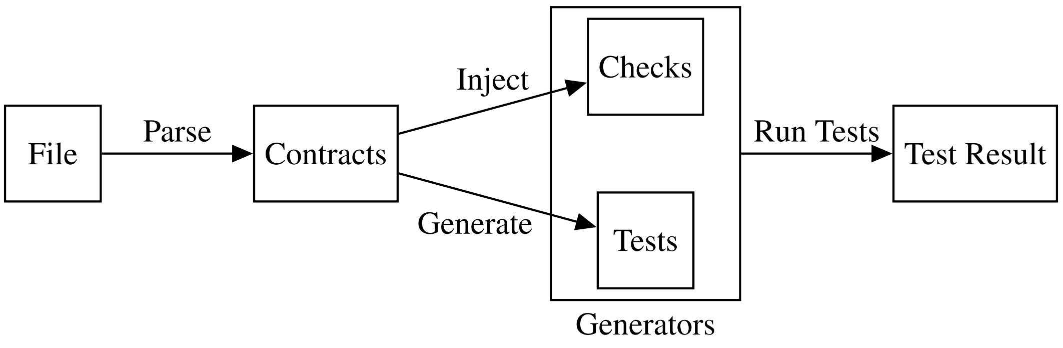


Figure 3.1: The flow of DropShadow where each square is an object or component. First, we have a ”File” including functions and contracts, which are parsed. These contracts are injected into the functions corresponding to the pre-, post-condition, and hyperproperty ”Checks”. Based on the same contracts, property-based ”Tests” are generated with corresponding termination criteria. Both the checks and tests require ”Generators” responsible for input generation and inclusion checks. When the ”Checks” have been injected and the ”Tests” are generated, we run the tests and receive a ”Test Result”.

Generators: DropShadow uses a composable architecture that allows for complex and custom generation of inputs. Its contracts, which are annotated as function documentation, are used to specify the valid inputs. Pre-conditions on the inputs are inclusion checks in the generator, and post-conditions are assertions. The contracts are written as correct Go code in the contracts, provided the generators are inserted with the ”pkg.” prefix.
Parse: DropShadow utilise Go’s native parser, `go/parser`, to parse Go source files containing the contracts. The parsing results in an Abstract Syntax Tree (AST), which is then traversed using Go’s native AST, `go/ast`, to build actual instances of the contracts specified in the source file. The results from parsing a file are the actual contracts associated with annotated functions,

[1]The name DropShadow originates from a visual interpretation of contracts as drop shadows for the functions.

including all metadata.
Checks: When the contracts of a source file have been constructed, DropShadow injects code according to the specifications of the contracts. The original signature of the function is preserved to ensure correct linking for callers. The injection of the contract is performed by adding nodes to the AST, which are responsible for all the logic of trace- and hyperproperties, as well as handling certain intricacies.
Tests: When contracts have been injected into the associated functions, DropShadow then constructs a test file containing generated property-based test cases as specified by the contracts. Post-conditions, in contrast to pre-conditions, are optional, and only contracts with post-conditions will have a corresponding test generated. DropShadow then automatically handles the execution of the generated test cases. After this, the test file is deleted and the source file is reverted to its original state.

With this summary of DropShadow the following sections will dive into each component and describe the design in more detail as well as describe how the theory introduced throughout the report has been implemented and applied practically.

3.1 Input Generation

Input generation is very useful for testing hyperproperties and a cornerstone of property-based testing, a methodology that shifts the focus from testing specific instances to general properties. Go already has a native fuzzer supporting some types, but not all, to be generated automatically [14]. Our contribution in this regard is a suite of generators for more types and custom types, as well as ways to work with arrays adhering to their generation requirements.

QuickCheck [4] is a well-known tool in the space of property-based testing, and DropShadow's input generation is similar in many ways - not least due to the minimalistic approach (what QuickCheck calls *test data generators* is called *input generators* in DropShadow). The main difference between the implementations is DropShadow's inclusion checking, which QuickCheck does not enforce all generators to have. DropShadow requires inclusion checks, as the generators are used to check pre-conditions, whereas QuickCheck only uses them for generational purposes without the function's contractual obligations on the inputs. Thereby, they do not aim for self-checking programs, but rather to randomly test properties of functions. Like QuickCheck, DropShadow allows the custom creation of generators such that users can control the generation of inputs. DropShadow also supports the `Any` generator, which is similar to `Arbritary` in QuickCheck.

In traditional unit testing, developers write tests for predetermined inputs and check the correctness of the outputs - usually structured like Arrange, Act, and Assert. This method, while useful, is limited by the developer's ability to anticipate problematic inputs, edge cases and boundary values while considering positive and negative test cases. Property-based testing addresses this limitation by defining properties, and general rules a function or system should adhere to, and then automatically generate inputs.

The efficacy of property-based testing hinges on the ability to produce a comprehensive set of test inputs, spanning the property's input space, without wasting resources generating inputs outside of it. DropShadow introduces the concept of generators to define, among other things, the input space for generation. Multiple generators can be used to specify a property reflecting a region of the input space (see Section 3.2). Existing property-based testing tools aim to enable developers to utilise the fuzzer and simply skip inputs outside the property, which in some cases is very inefficient. In addition, not all types are supported, creating a significant challenge for users who need to convert the fuzz input into the required type.

```go
type Generator[T any] interface {
  Contains(value T) bool
  NextValue() T
}
```

Listing 3.1: The interface required for all generators.

In Listing 3.1 the interface required by all generators is presented. This simple interface enables generation and inclusion checks. Without this, the input generation required for testing hyperproperties and test case generation would be difficult. It allows for custom generators made by users, numeric intervals, array generation, and much more.

```go
type SizedGenerator[T any] interface {
  Generator[T]
  Size() uint64
}
```

Listing 3.2: The interface required for all sized generators.

At times, it is desirable to combine regions which, otherwise, only express a single partition to instead express a set of regions. DropShadow supports combining some generators through the use of `DisjointUnion`. However, to not favour some inputs more than others we require the generators to have a notion of size corresponding to the amount of different values it can generate. In Listing 3.2 the interface definition for these generators can be seen. Fundamentally they simply extend `Generator` with a function `Size`. This function returns, as a `uint64`, the number of values it can distinctly generate. For user-defined sized generators they might find cases where the definition of size in some way is altered to a more general notion of weight. This would allow custom distributions over multiple generators. The way `DisjointUnion` works is by returning a sized generator composed of multiple sized generators. The total size is then the summation of all sizes. Sampling is then done by weighted random over the sized generators, and an inclusion check is performed by just finding one generator able to generate the value.

3.1.1 Numeric Intervals

The usage of numbers is often an essential requirement for most software. For this reason, Drop-Shadow supports a wide range of generators representing numeric intervals. Fundamentally, an interval is just a `min` and `max` value of a generic numeric type and two booleans describing the ends as either open (exclusive) or closed (inclusive). For instance, a fully closed generator over some number can be constructed with the use of `Inclusive`, and `Exclusive` for fully opened generators. In addition, it can be more expressive to describe bounds which range from one point and continue to the extreme in one direction. To support this we have `LT` ($<$), `LE` ($\leq$), `GT` ($>$), and `GE` ($\geq$).

```
type NumericInterval[T Number] struct {
  min     T
  max     T
  openMin bool
  openMax bool
}
```

Listing 3.3: The `NumericInterval` struct representing a range of numeric values in DropShadow.

In Listing 3.3, the struct for the numeric intervals is presented. This struct implements the `SizedGenerator` interface to be considered a valid sized generator. To do so, the numeric intervals have both a generator function (`NextValue`), inclusion check (`Contains`), and size (`Size`). The `Contains` method defined on `NumericalInterval` checks if a given value falls within the interval, taking into account whether the ends are open or closed (Listing 3.4).

```
func (interval *NumericInterval[T]) Contains(value T) bool {
    if !interval.openMin && !interval.openMax {
        return value >= interval.min && value <= interval.max
    }
    ...
 }
```

Listing 3.4: Excerpt from the `Contains` method.

```go
func (interval *NumericInterval[T]) NextValue() (value T) {
  switch any(value).(type) {
  case float64:
    value = T(Float64InRange(float64(interval.min), float64(interval.max)))
  case int:
    min, max := interval.integerBounds()
    value = T(SignedInRange(int(min), int(max)))
    ...
  }

  if !interval.Contains(value) {
    return interval.NextValue()
  }
  return value
}
```

Listing 3.5: Excerpt from the `NextValue` method.

The `NextValue` method generates a pseudo-random value within the interval, catering to the different supported numeric types. In Listing 3.5 our generation of `float64` and `int` can be seen. To generate the values, the global `rand` instance is used from the `rand` package by the functions `Float64InRange` and `SignedInRange`. For floats, we over-approximate the interval assuming both ends are closed. This can be problematic for partially or completely open intervals. However, the function makes an inclusion check on the generated value and if `false` is returned, a new value is generated until one is found, that falls within the specification.

3.1.2 Array Generation

Arrays are often used in software as sequences or collections of elements. However, its generation is much more complex than numeric intervals. First, one should be able to generate an array of any underlying type, which also implies multidimensional arrays. Second, the construction of array elements should, in some cases, have access to the previous elements. Essentially, this would allow for the construction of sorted arrays.

Go already has a native fuzzer, but it is severely restricted by its ability to only generate one-dimensional byte arrays. To resolve this issue, we have constructed two generic generators which can be extended by the users. The first generator generates all values from a separate generator, without considering the previous generated elements. The second generates elements based on preceding elements through the `step` function. Both are not sized, making `DisjointUnion` over these array generators infeasible. This restriction originates from the size growth of arrays, which can quickly become very large. In the cases where sized generators of arrays are required, users must create a custom generator for it.

```
type ArrayGenerator[S ~[]E, E any] struct {
  factory       Generator[E]
  configuration ArraysConfiguration
}
```

Listing 3.6: The `ArrayGenerator` generator.

In Listing 3.6 we see the generator for an array where all elements are created from `factory` generator. The `configuration` contains metadata for the construction of the array, such as bounds (which itself is also a generator). To support the configuration of arrays we have utilised the configuration pattern, which follows the open-closed principle, ensuring an extensible configuration for arrays, that allows for further addition. An inclusion check of an array is performed by checking each element to see if the generator has been able to generate it.

```
type ArrayStepGenerator[S ~[]E, E any] struct {
  step          func(array S, index int) Generator[E]
  configuration ArraysConfiguration
}
```

Listing 3.7: The `ArrayStepGenerator` generator.

In Listing 3.7, the generator for what we call a step generator can be seen. The generator requires a `step` function which deterministically creates a generator for the array elements provided the previous elements in the current state of the array (it has to be deterministic as it is also used for inclusion checks). This generator is useful in cases where a sorted array should be generated.

The `step` function takes an index of the current element of the array, from which the factory can consider all previous elements. Calling `step` yields a generator, which in turn generates the preceding element. Inclusion is done by checking, provided the current array from the index, the returned generator includes the next element.

3.1.3 Any Generation

Some functions should be able to accept all inputs. But some inputs, when generating them, can yield incredibly large objects. For example, consider a generator designed for arrays with an unconstrained length; such a generator could potentially produce an array as large as the maximum value of an `int32`. Moreover - functions might accept all numeric values of a specific type. To facilitate this, the `AnyNumber` function is used. It creates a closed interval bounded by the minimum and maximum values of the specified numeric type.

```
func AnyNumber[T Number]() *NumericInterval[T] {
  return Inclusive[T](MinOf[T](), MaxOf[T]())
}
```

Listing 3.8: The `AnyNumber` generator.

The `AnyGenerator` is a special unsized generator which returns `true` for all inclusion checks but panics when attempting to generate a value. The reason for this is, that construction of any objects (objects implementing `interface{}`) can only be done for its zero value. We believe the zero value does not necessarily fit well with the generator's name. A generator like `AnyGenerator`, which returns true for all inclusion checks and cannot generate a value, is useful for cases where one needs a catch-all pre-condition. Another argument for its need is, that not all regions should have a test case; the generation of arbitrary arrays, for example, can yield very large arrays. Essentially, `Any` supports a general usage pattern we identified for DropShadow.

```go
func Any[T any]() *AnyGenerator[T] {
  return &AnyGenerator[T]{}
}
```

Listing 3.9: The `Any` generator.

To ensure every type required by users can be generated, users can create their own generators and call `Custom` with the generator as a parameter. DropShadow has the requirement that all pre-conditions are functions from the `pkg` package returning a generator. Therefore, to support custom generators (a function that essentially just returns the passed generator to `Custom`), this function seems like the only approach for user-defined generators.

```go
func Custom[T any, G Generator[T]](generator G) G {
  return generator
}
```

Listing 3.10: The `Custom` generator.

In conclusion, input generation is a pivotal feature of DropShadow, driving the tool's ability to perform property-based testing, hence uncovering potential issues that might not be evident with traditional example-based testing and fuzz testing. By allowing developers to specify input regions, DropShadow ensures that a wide array of inputs is automatically generated and tested, fostering confidence in the reliability and correctness of Go programs. However, the effectiveness of this approach hinges on the careful definition of regions by the developer. Incorrectly specified regions could either miss critical test cases or generate irrelevant inputs, thus impacting the thoroughness and efficiency of tests.

3.2 Contract

A contract in DropShadow allows a developer to write specifications for a function to describe its behaviour. Contracts can be used to check compliance with the function's pre- and post-conditions. This is done through an automatic process, where assertions are derived from the contract and arbitrary data is generated to match the specification to conduct property-based testing as well as testing for hyperproperties. In addition to this, our separation of specifications into (potentially overlapping) named regions enables future work regarding debugging the path to a failed contract just by following a sequence of sets of region names. Through named regions, developers can abstract potentially complex region logic into everyday language.

In Listing 3.11 a contract can be seen. This contract is formulated with Go code in the function's documentation. Using Go reduces the mental overhead of the developer when writing contracts and also enables a more straightforward conversion between them and the property-based tests derived from them.

```go
// region: Red
// config: 200s, 100it
// assume: x = Inclusive[int](1, 3)
// assume: y = Interval[int](1, 6, true, false)
// expect: ret == "red"
// hyper:
// assume: x_p = x
// assume: y_p = y
// expect: ret_p == ret
// region: Green
// config: 200s, 100it
// assume: x = Inclusive[int](7, 8)
// assume: y = Inclusive[int](6, 8)
// expect: ret == "green"
func Rectangle(x, y int) string {
    if x >= 1 && x <= 3 && y > 1 && y <= 6 {
        return "red"
    }
    if x >= 7 && x <= 8 && y >= 6 && y <= 8 {
        return "green"
    }
    panic("Unknown rectangle")
}
```

Listing 3.11: A function `Rectangle` with two named regions - one including a hyperproperty returning what color the rectangle should be. In the red region `y` uses the `Interval` function with `true` and `false`. `true` makes the lower bound open and `false` the upper bound closed.

The DropShadow contract is composed of one or more regions, which are specified using the keyword `region` followed by an optional region name. A region must have a pre-condition (`assume`), for each of its formal parameters, which yields a generator. An arbitrary amount of post-conditions (`expect`) is allowed. The order of the assumes is significant as the actual parameters are inserted in that order, resulting in the first `assume` being the first parameter. If the contract has no post-condition, then no test will be generated.

Each region can have at most one hyperproperty defined by the `hyper` keyword. The structure of the hyperproperty is not entirely the same as `region`. The pre-conditions of a hyperproperty must be a value and not a generator. The post-condition for hyperproperties is formulated the same way as with `region`. In addition, `config` is used to specify test execution termination criteria, which can either be a specific time and/or specific iterations (invocations) of the test target. In the following, we will describe the components of a DropShadow contract.

`assume` and `expect` both have a left-hand side (lhs) and right-hand side (rhs) centred from the equal sign. With `assume` in `region`, the lhs is the identifier of the formal parameter and the rhs is the function from `pkg` that is called, yielding a generator. The function's valid inputs are then the set of all regions which pass the generators' inclusion check. Only if an input is in no region will the pre-condition fail.

With `expect`, there is no lhs or rhs. Everything after the colon is the boolean expression, tested when checking the post-condition for all regions of which the inputs are a part. If it results in any one of the post-conditions being false, there is a breach of contract. Regarding hyperproperties, `assume` has primed (... _p) variables for parameters. Here, the actual parameters of the invocation returning the primed return value are the sequence, the assumes are specified in - there is no correlation between the identifiers of primed parameters and the identifiers of the function. Only the order of the assumes determines the placement of the actual parameters.

3.2.1 Region

The contract in Listing 3.11 shows an example of a contract containing two regions named `red` and `green`. A region in the DropShadow contract describes an area in the input space. It specifies the allowed area, in which the generators generate values. A visualisation of the regions specified in Listing 3.11 can be seen in Figure 3.2.

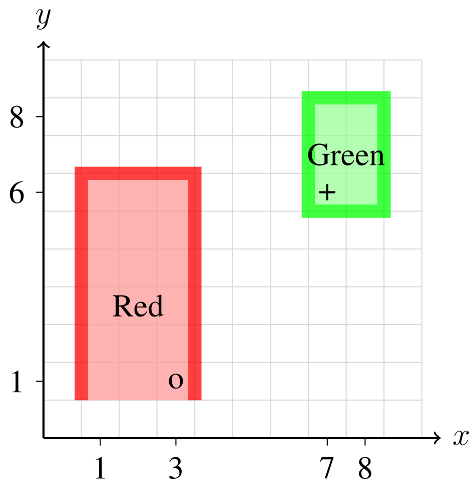


Figure 3.2: 2 regions red, and green. Green is fully closed, and red is closed on x but half open on y. "o" is on $(3, 1)$ and not included in red as it is open. "+" is on $(7, 6)$ and included in green.

The function in Listing 3.11 has two parameters, `x` and `y`. Each of the two regions are drawn with the color corresponding to the region's name and `expect`. For simplicity, we chose numeric typed parameters. However, more complex and user defined types can be used.

The format of contracts in DropShadow has been presented as well as our approach to input generation with the use of generators. However, there are notable drawbacks to writing contracts in function documentation, that ought to be mentioned. First of all, The source files may appear cluttered due to overlapping specifications. Additionally, there is a learning curve involved, and finally a specific, necessary generator might be required, which could be difficult to implement.

3.3 Parsing

In [15] the grammar for Go is presented in a variant of Extended Backus-Naur Form, where the empty string is omitted and iteration constructs can be used. In Listing 3.12 the grammar for DropShadow's contracts is shown. To parse Go and retrieve the functions' documentation, we used their open-source parser and AST struct. Our parsing does not concern the parsing of Go itself, only the contracts. Using the same variant, we have described the contract grammar below. Comments in Go are lexical constructs without a production we can extend. There are two types of comments: line comments which start with "//" and stop at the end of the line, and general comments which start with "/*" and stop at the first subsequent "*/". Our contracts are constructed by a sequence of line comments as per the godoc best practices of function and method documentation [13].

```
Contract    =  ( Obligations | Region ) { Region } .
Region      = "//" "region:" [ identifier { identifier } ] Obligations .
Obligations = { Assume | Expect | Config } [ Hyper ] .
Config      = "//" "config:" Termination { "," Termination } .
Termination = ( Expression "it" ) | ( Expression "s" ) .
Hyper       = "//" "hyper:" { Assume | Expect } .
Assume      = "//" "assume:" Expression .
Expect      = "//" "expect:" Expression .
```

Listing 3.12: EBNF for DropShadow contracts.

Within our grammar, we have inserted "//" to denote the start of the line comment and all productions starting with it, assuming it ends at the end of the line as defined by line comments in Go. From the "//" to the following string, such as "hyper", there can be arbitrarily much whitespace, which is accounted for by separating the two strings. In Go's specification, function documentation is not a part of its grammar, but it is required to immediately precede the function definition[2]. However, the resulting AST node for a function definition does include the documentation. Our parser essentially works on each line comment of the documentation, checking what prefix it begins with: "region:", "hyper:", "assume:", or "expect:". Based on this prefix, we then know what production to parse. In the case of "Termination" we split on the separator and check the suffixes. From the specification, it is also possible for a region or hyper to have no assumption or expectation. Even though this is possible, it is not necessarily useful.

[2]https://go.dev/ref/spec#Function_types

The following is a brief textual presentation of the grammar.

- *Expect:* the post-condition with a Go ”Expression”. Its grammar can be found in the cited source.
- *Assume:* the pre-condition with a Go ”Expression”.
- *Hyper:* the pre- and post-conditions for a hyperproperty.
- *Termination:* a termination criterion is either an expression evaluating a number for the number of iterations or seconds.
- *Config:* the configuration of the termination criteria for the region's generated test.
- *Obligations:* the pre-conditions (assumes), and the post-conditions (expectations) followed by an optional hyperproperty.
- *Region:* named or not by one or more identifiers, followed by the region's obligations. The ”identifier” rule refers to Go's own identifier. If multiple identifiers are present then they are combined with ”_” making them a valid Go identifier which can be used to construct the identifiers for their respective tests.
- *Contract:* the full function documentation with either an explicit region specific or just contractual obligations. This is followed by zero or more regions.

In summary, the grammar is more formally specified for the contracts. We have presented our parsing approach as well as textual descriptions. We will now continue to contract injection which concerns processing the parsed contracts and inserting them into the function.

3.4 Contract Injection

So far, contracts as well as regions have been presented. Now we will cover the injection of checks. This injection moves the contractual specification into the function, which forces callers to comply with the callee's contract. In this section, we will cover the implementation details and complexities.

In general, the approach works on the AST of the contract-annotated function. First, the function is taken and declared as a function named `wrap` within itself; second, we inject the region checks; third, the post-condition checks; fourth, the hyperproperty test is injected. At last, a new return statement is inserted, which returns the value from calling `wrap`. In addition to this, we also have to ensure all required packages are imported, e.g. `pkg` with the generators.

3.4.1 Wrap

Several challenges are solved by wrapping the function and declaring it within itself. If not, injecting post-conditions and hyperproperty tests would be more complex with multiple return statements. We could have used the `defer` functionality in Go to call the required logic right before each return statement. However, by wrapping the function, we separate the scopes, generally making it simpler to programmatically generate variable identifiers which do not collide with others. Most likely, wrapping the function introduces some performance overhead, but we view this as insignificant in most situations. Moreover, by wrapping we can create the variable `ret` for the returned value of `wrap`, which can be referenced in the contracts. Without this, we would require named returns, as implemented by other solutions. However, this approach would also alter how programmers write their functions.

```go
func Rectangle(x, y int) string {
    ...
    wrap := func(x, y int) string {
        if x >= 1 && x <= 3 && y > 1 && y <= 6 {
            return "red"
        }
        if x >= 7 && x <= 8 && y >= 6 && y <= 8 {
            return "green"
        }
        panic("Unknown rectangle")
    }
    ret := wrap(x, y)
    ...
    return ret
}
```

Listing 3.13: Wrapping of the Rectangle function into wrap.

Wrapping the function is a simple yet very effective way to solve the challenges inherent in contract injection. In Listing 3.13 we can see an example, where the wrap is declared with the same original signature and body. Beneath it, `ret` is declared to be the returned value of `wrap` called. Notice how the function signature remains unchanged. This ensures the same callers can still call the function.

3.4.2 Assume

To ensure all calls to a function complies with the contractual specification, inputs are initially checked against the pre-conditions. As previously mentioned, the assumptions are calls to functions in `pkg` returning a generator. Lines 2-5 in Listing 3.14 display the generators for the input arguments `x` and `y`, which are the specified assumes in the contract. The letter *s* is used as an appended suffix to distinguish between the generators and their region. These generators define the allowed range, the input argument is allowed to be within. With DropShadow being a proof-of-concept we expect this approach to be sufficient. However, for functions with, e.g., the formal parameters `x int, xs []int` the appending of s's to the parameter names would cause problems.

```go
1  func Rectangle(x, y int) string {
2      xs := pkg.Inclusive[int](1, 3)
3      ys := pkg.Interval[int](1, 6, true, false)
4      xss := pkg.Inclusive[int](7, 8)
5      yss := pkg.Inclusive[int](6, 8)
6
7      fRegion := func(x, y int) []int {
8          ret := make([]int, 0)
9          if xs.Contains(x) && ys.Contains(y) {
10             ret = append(ret, 0)
11         }
12         if xss.Contains(x) && yss.Contains(y) {
13             ret = append(ret, 1)
14         }
15         return ret
16     }
17
18     regions := fRegion(x, y)
19     if len(regions) == 0 {
20         os.Exit(1)
21     }
22     ...
23 }
```

Listing 3.14: The code inserted by DropShadow to account for assumes in a contract.

Lines 7-16 show the function `fRegion` used to determine which regions the parameters would fall into, manifested by the inclusion checks on the generators (lines 9 and 12). `fRegion` returns an integer array of all the indices of the regions. Even though this function is only used on line 18 and hence could be inlined, our way of organising it keeps the logic separated.

On line 19, an emptiness check is performed on the region set. The absence of regions indicates the input values are not in any allowed area. Thus, a violation of the contract has occurred (the pre-conditions are not met) and an `os.Exit(1)` is hence invoked in line 20. This will terminate the execution of the function.

3.4.3 Expect

Once the wrapped function body has been executed to obtain the resulting value `ret`, and the input has been validated against the assumes, we must check the result against the post-condition specified in `expect`. Hence, the `expect` needs to be inserted as assertions in the code. These expects can be different from `region` to `region` - therefore, we must match the post-condition with the correct `region`. To account for this, we make use of a switch statement with a case for every such region. This structure can be seen in Listing 3.15, which is the code inserted by DropShadow to implement the expects defined in Listing 3.11. Using this structure, we can add more cases as the number of regions grows in the contract.

```go
func Rectangle(x, y int) string {
    ...
    for _, region := range regions {
        switch region {
        case 0:
            if !(ret == "red") {
                os.Exit(1)
            }
        case 1:
            if !(ret == "green") {
                os.Exit(1)
            }
        }
    }
    ...
}
```

Listing 3.15: The injected assertions by DropShadow which accounts for the expects defined in contracts.

An `expect` is formulated as a boolean expression in Go code and can thus be negated and inserted directly into an `if`-statement to check for violations. As the number of expects increases within a region, more `if`-statements are added. As is the case with assumes, an `expect`-violation also invokes `os.Exit(1)`.

3.4.4 Hyper

For contracts involving hyperproperties, we utilise sequential self-composition to reduce them. The result of DropShadow injecting the hyperproperty in Listing 3.11 can be seen in Listing 3.16. Similar to `expect`, a hyperproperty can be defined for each region. So, we employ a similar structure to `expect`, using switch cases to effectively manage various regions. However, in this example, only one hyperproperty has been specified in the *Red* region. Crucially, another challenge addressed by wrapping is, that it allows us to invoke it when testing the hyperproperty. Note the assumptions and expectations are treated as expressions without creating a new generator.

```go
 1 func Rectangle(x, y int) string {
 2     ...
 3     for _, region := range regions {
 4         switch region {
 5         case 0:
 6             x_p := x
 7             y_p := y
 8             ret_p := wrap(x_p, y_p)
 9             if !(ret_p == ret) {
10                 os.Exit(1)
11             }
12         }
13     }
14     return ret
15 }
```

Listing 3.16: The code inserted by DropShadow to check for hyperproperties.

The primed input variables in lines 6-7 are used for the second execution of the program to obtain another execution trace. As the original body of the function is encapsulated in the wrap function, we can thus execute the wrap function once again - now with the primed input variables. With two different execution traces of the wrap function, the results can then be asserted, as can be seen in line 9.

3.5 Test Generation

The test generation phase in DropShadow involves the creation of test cases for the functions which have specified contracts made by the developer. DropShadow's test generation mechanism operates post the contract injection phase, where Go functions are already augmented with contract-based assertions. Utilising these injected contracts, DropShadow fabricates a dedicated test file containing a series of test cases, where a case is created for each region specified in a contract.

These test cases run the functions to conduct the property-based testing. Since the responsibility for handling breaches of contract is managed within the function under test itself, DropShadow adopts an Arrange-Act structure. This contrasts with the traditional Arrange-Act-Assert approach, where assertions are made in the test case. In other words, one can view a test case in DropShadow more as a driver to run the function under test.

The construction of test cases is segmented into the following steps:

- **Generator initialisation**: for each pre-condition within a contract region, a corresponding input generator is initialised. These generators are tasked with producing diverse input values conforming to the assumptions stated in the contract.

- **Test execution criteria**: the test framework creates execution criteria based on the contract's termination specifications, which can be time-based, iteration-based, or a combination of both. In case no conditions have been specified, DropShadow will use the default settings, which run the tests for 10 seconds with unlimited iterations.

- **Function invocation**: the target function is invoked with inputs created by the generators. This creation is the Arrange, and the invocation is the Act.

Applying these steps on the contract of the *Rectangle* function specified in Listing 3.11 will result in the generation of two test cases, as two regions are specified within the contract. As an example Listing 3.17 shows one of the generated test cases.

```go
func Test_Rectangle_Red_Property(t *testing.T) {
  xGenerator := pkg.Inclusive[int](1, 3)
  yGenerator := pkg.Interval[int](1, 6, true, false)
  iterations, iterationLimit := 0, 100
  startTime, timeLimit := time.Now(), 200*time.Second
  for {
    x := xGenerator.NextValue()
    y := yGenerator.NextValue()
    Rectangle(x, y)
    iterations++
    if iterations >= iterationLimit {
      return
    }
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
```

Listing 3.17: The automatically generated test case for the Red region.

Notice how the test case does not contain assertions to evaluate the test case, which is usually done by asserting the function's output against an expected one, as seen in the Arrange-Act-Assert paradigm. Furthermore, the test case does not contain mechanisms to fail the test case. The test case will only end its execution either when the termination conditions are met, or if the function under test encounters a breach of contract based on the input provided by the test case. In this case, the function invokes an `os.exit(1)`, as explained in earlier sections. In Section 5.1 we will discuss the implications and alternatives to this approach.

Chapter 4

Evaluation

In this chapter, we will present an evaluation of DropShadow with a focus on showing the capabilities of contracts, trace properties, and hyperproperties. First, we will compare how the contracts in DropShadow are written with the tool gocontracts, and how our contracts support property-based testing. Second, we will show that both DropShadow and gocontracts can describe trace properties.

4.1 Contracts

When comparing the contracts between DropShadow and gocontracts we will look at how pre- and post-conditions are formulated. When the two approaches have been presented, we will consider how a developer would utilise the contracts. By that, we show how the structure of DropShadow's contracts is sufficient to support the automatic generation of property-based tests whereas gocontracts are not. To find suitable tools for Go to compare DropShadow with, we looked on GitHub for projects under ”dbc”, and ”design by contract” topics as well as similar searches in the same domain. In descending order of GitHub stars, we have identified the following projects to name a few: pact-go [12], Parquery's gocontracts [28], grpc-go-contracts [20] and dbc4go [3].

What we found was that most projects are Application Programming Interfaces (APIs) for writing assertions or formulating pre- and post-conditions as assertions in the code. Only Parquery's gocontracts and dbc4go enabled users to write contracts in the documentation for functions and inject them. Since this injection of contracts is of central importance in DropShadow, we will mainly centre our evaluation around comparing DropShadow and gocontracts. The other tools mostly resemble APIs for programming with assertion. This can be used for manual testing of hyperproperties, but rather we are interested in the automatisation. In short, the tool's focuses are: first, pact-go is a contract testing tool aimed towards microservice-oriented architectures with RESTfull HTTP APIs [12]. Second, Parquery's gocontracts supports contractual design for Go with pre- and post-conditions and distinguishes between in testing and in production contracts [28]. Third, grpc-go-contracts wraps around GRPC checking the requests for pre-conditions and the responses for post-conditions [20]. Fourth, dbc4go like gocontracts enables contractual specification in function documentation which is inserted in the code [3].

From this, we can see that Parquery's gocontracts and dbc4go are the only ones support contracts as documentation. The main difference between the two is that dbc4go supports invariants and `@import` to specify additional required imports used in the contract. However, gocontracts implementation of `preamble` makes it a better fit for showing another approach of doing sequential self-composition, which is another reason for focusing on it.

For now, the comparison will focus on the `Absolute` function. This function, a simple yet fundamental operation, provides a good basis for illustrating the differences in expressiveness, flexibility, and syntactical requirements of the two tools (see Listing 4.1 and Listing 4.2). These aspects were chosen because we found them to be the most comparatively unique and effectual for their users. But, before diving into these aspects we will describe the two examples in more detail to highlight distinctions and capabilities.

```go
// region: Zero
// config: 1it
// assume: value = Constant[float64](0)
// expect: ret == 0
// region: Negative values
// config: 10s
// assume: value = LT[float64](0)
// expect: ret == -value
// region: Positive values
// config: 10s
// assume: value = GT[float64](0)
// expect: ret == value
// region: All
// config: 100it
// assume: value = AnyNumber[float64]()
// expect: ret >= 0
func Absolute(value float64) float64 {
    if value < 0 {
        return -value
    }
  return value
}
```

Listing 4.1: Contract for Absolute expressed with DropShadow.

In Listing 4.1 four regions are defined `Zero`, `Negative values`, `Positive values`, and `All`. The injected contract can be seen in Listing A.1. The `Zero` region has a `config` with `1it` meaning that its generated test will only be run once. Whereas `All` is run 100 times and `Negative values` is run for 10 seconds.

```go
// Absolute ensures:
//  * value < 0 && result == -value
//  * value > 0 && result == value
//  * value == 0 && result == 0
//  * result >= 0
func Absolute(value float64) (result float64) {
    if value < 0 {
        result = -value
    } else {
        result = value
    }
  return
}
```

Listing 4.2: Contract for `Absolute` expressed with gocontracts.

In Listing 4.2 the same four regions have been specified with gocontracts. The injected contracts can be seen in Listing A.3. We have found the following distinctions between DropShadow and gocontracts:

- gocontracts interpret functions with no pre-condition to accept every input. If there are pre-conditions then gocontracts support a ”require” block. On the other hand, DropShadow requires all parameters to be specified with an assume in all regions. This ensures a generator for each parameter can be constructed for the property-based tests.

- DropShadow requires the pre-conditions to be generators from `pkg`, whereas gocontracts only requires a boolean expression to describe input validity. Generating inputs for functions in DropShadow is done by calling a function on the generator for the parameter. gocontracts, on the other hand, would require some additional work on filtering invalid inputs and generating values. Additionally, DropShadow supports the construction of arrays. Whereas gocontracts would find even more difficulty writing tests with random arrays as even the built-in fuzzer (go-fuzzer) does not support the construction of such types. In addition, DropShadow can be extended to custom types where users can define generators for structs and such by composing existing generators.

These distinctions clearly show how the tools are meant for different things but with the same contractual approach.

Expressivenes and clarity: DropShadow utilises a region-based approach to define the input space for testing. Each region, such as `Zero`, `Negative values`, and `Positive values`, is explicitly declared with assumptions about input values and expectations for the function’s output. gocontracts, on the other hand, use a more concise syntax to express pre- and post-conditions within a single comment block. The conditions are written in an assertion-like style, focusing on the relationship between inputs (`value`) and the named return variable (`result`).

Flexibility in test configuration: DropShadow's contracts allow for fine-tuned test configurations within each region, using directives like `config` to specify test termination criteria. This level of control is beneficial for tailoring the testing process to different scenarios, such as allocating more resources to critical or complex regions. gocontracts does not include an explicit mechanism for such configurations within the contract syntax, simply because this tool does not generate and evaluate test cases.

Function syntax and return values: DropShadow's approach does not require named return variables in the function definition, as it wraps the function under test and captures its return value in a variable named `ret` (The primed return value is stored in `ret_p`). This means that developers do not need to alter the function signature solely for testing purposes, but do have to remember them when writing the contracts. In contrast, gocontracts require named return variables for the contract to reference the function's output correctly (the `result` variable).

In conclusion, DropShadow offers a detailed and configurable approach to defining test properties and regions, which can be particularly beneficial for comprehensive testing scenarios. gocontracts provides a concise and straightforward way to express function contracts, favouring simplicity and readability - especially in a simple case such as the `Absolute` function.

4.2 Trace Properties

In this section, we will evaluate DropShadow's approach to property-based testing focusing on trace properties - a single execution. Unlike existing tools such as *go/quick* [16], *rapid* [30], and *gopter* [22], which requires the developer to manually write test cases, DropShadow automates this process by generating test cases from the contract specifications. We chose to compare these tools as they represent a popular choice among developers and `go/quick` which is a part of the standard library. The evaluation will focus on the effort required for test case creation, how well inputs are covered, and how test cases are structured to handle different properties across the tools.

We will use the `GetDiscount` function seen in Listing 4.3 to evaluate the tools. This function was chosen due to its many possible paths of execution. The nature of which, requires a thorough contractual specification for the function. It takes two integer parameters, a postal code and a month, and then returns an integer representing a discount. The function is designed to check for three properties:

- A validity property, which ensures all input outside the valid postal code and month ranges return a default value
- A seasonal property, which ensures certain postal codes receive a specific discount during summer months.

- A constant activity property, which ensures certain postal codes always return the same discount regardless of the month.

Danish postal codes typically range from 1000 to 9999 but for simplicity, we will not account for special and non-existing postal codes within this range.

```go
 1 func GetDiscount(postalCode int, month int) int {
 2   if postalCode < 1000 || postalCode > 9999 || month < 1 || month > 12 {
 3     return 0
 4   }
 5   switch {
 6   case postalCode >= 1000 && postalCode <= 1999:
 7     return 20 // Constant discount in core city area
 8   case postalCode >= 2000 && postalCode <= 3899:
 9     return 10 // default
10   case postalCode >= 3900 && postalCode <= 3999:
11     if month >= 6 && month <= 8 {
12       return 40 // Summer discount
13     }
14     return 10
15   case postalCode >= 4000 && postalCode <= 9999:
16     return 10
17   default:
18     return 0
19   }
20 }
```

Listing 4.3: Function used for conducting property-based testing.

Seen on line 2 is the valid Danish postal code and month range check. The switch statement on line 5 determines the different discounts based on the ranges of postal codes and months. We want to see how the different tools for property-based testing handle these discounts as they depend on both postal codes and months, essentially dividing the input space of valid inputs.

To show the differences in the structure of the test cases and their complexity, examples of test cases created using the tools *go/quick*, *rapid*, and *gopter* are provided in Appendix B. The examples follow the *arrange, act, assert* structure and highlight the manual effort required in defining the generators and properties for the test cases. Common for these tools is, that the properties of the function under test are expressed in the test cases rather than the function. They follow the pattern of defining the property to be tested, specifying the inputs within it, and then invoking the function under test to determine whether it is fulfilled. This could lead to maintainability issues, as each of the test cases would have to be rewritten in case the function under test is refactored and properties altered.

To understand how test cases are written using *go/quick*, consider Listing 4.4 which shows the test of the validity property. An anonymous function is defined on line 2, returning a boolean which is stored in the `property` variable to indicate whether the property is fulfilled. Within the body of the anonymous function, the function under test is invoked and the property specified. The actual result is then evaluated against the expected one, which can be seen on lines 5 and 7. With the property defined, the test is run on line 18 by invoking *quick.check()* with the property and a configuration defined on line 10. By default *go/quick* runs for 1000 iterations, but this is configurable. *quick.check()* calls `property` repeatedly, with arbitrary values for each argument passed to the function under test (`GetDiscount`). To communicate a violation of the property, `t.Error()` function from the go/testing library can be used.

```
1  func TestSeasonalDiscountPropertyQuick(t *testing.T) {
2    property := func(month, postalCode int) bool {
3      discount := GetDiscountOtherTools(postalCode, month)
4      if month >= 6 && month <= 8 {
5        return discount == 40 // Summer discount
6      }
7      return discount == 10
8    }
9
10   config := &quick.Config{
11     MaxCount: 1000,
12     Values: func(v []reflect.Value, r *rand.Rand) {
13       v[0] = reflect.ValueOf(1 + r.Intn(12)) // Month: [1; 12]
14       v[1] = reflect.ValueOf(3900 + r.Intn(100)) // Postal code: [3900; 3999]
15     },
16   }
17
18   if err := quick.Check(property, config); err != nil {
19     t.Error("Failed seasonal discount test:", err)
20   }
21 }
```

Listing 4.4: A go/quick test case for the seasonal discount property.

When writing test cases using rapid, the `rapid.Check()` function encapsulates the whole test and is used to execute the property test seen in Listing 4.5. The first parameter is a pointer to `testing.T` which allows rapid to integrate with the standard Go testing library. The second parameter is the anonymous function, which defines the test and properties specified. On line 3 and 4 *rapid.IntRange* is used to define integer ranges for the generators. By invoking `Draw()`, rapid will randomly produce values from the generator which fall within the specified range. The function under test is invoked on line 7 with the produced values, and the property is evaluated on line 9. The `rapid.Check()` fails the current test in case it encounters a panic or a call to `t.Fatalf()` (line 10) or similar calls from the standard testing library.

```go
func TestSeasonalDiscountPropertyRapid(t *testing.T) {
  rapid.Check(t, func(t *rapid.T) {
    postalCode := rapid.IntRange(3900, 3999).Draw(t, "postalCode")
    month := rapid.IntRange(1, 12).Draw(t, "month")
    isSummer := month >= 6 && month <= 8

    discount := GetDiscount(postalCode, month)

    if (isSummer && discount != 40) || (!isSummer && discount != 10) {
      t.Fatalf("Seasonal discount property violation")
    }
  })
}
```

Listing 4.5: A rapid test case for the seasonal discount property.

A property test written in gopter can be seen in Listing 4.6, where in line 2 `gopter.NewProperties()` such a test is created. It is parameterised with the default settings, but it can also be configured. The property definition is defined on line 4, where a name is added to the test; the `prop.ForAll()` specifies the property which should hold for all values generated by the generators. It takes an anonymous function (line 5) and two generators as arguments (line 12 and 13). Similar to *go/quick* and rapid, the anonymous function is used to specify the logic for the property test, which invokes the function under test and evaluates the specified property. Moreover, the generator of gopter can specify ranges as seen in rapid, which restricts the generator to only produce values within a given range. To communicate property violations, gopter also utilises the standard testing library like the other tools. However, this is abstracted away from the developer and is handled within the gopter library.

```go
func TestSeasonalDiscountPropertyGopter(t *testing.T) {
  properties := gopter.NewProperties(gopter.DefaultTestParameters())

  properties.Property("Seasonal discount property", prop.ForAll(
    func(postalCode int, month int) bool {
      discount := GetDiscountOtherTools(postalCode, month)
      if month >= 6 && month <= 8 { // Summer discount
        return discount == 40
      }
      return discount == 10
    },
    gen.IntRange(3900, 3999),
    gen.IntRange(1, 12),
  ))

  properties.TestingRun(t)
}
```

Listing 4.6: A gopter test case for the seasonal discount property.

From the examined tools we see, that the effort required to express the function specification is not just specifying input ranges - some setting up is also required. At first glance, `go/quick` might be the simplest to understand. However, its generational methods can be very inefficient, which is visible in the filtration of inputs. In contrast, the other tools can express more complex generators allowing for ranges to be specified: for example, gopter and quick uses reflection in their generators, whereas DropShadow uses parameterised types only introduced in the later version of Go (v. 1.18).

All the tools are similar in how tests are written - higher-order function encapsulating the test and in some way, either a boolean or calling a function announces whether a test failed. However, they differ much more in how they generate values. Quick and Gopter allows users to specify custom generators per parameter of the test function. Both heavily rely on reflection for runtime-type conversions. Rapid has moved the generation into the test and relies on type parameterisation on their generator eliminating the need to wrap generated values into a `relfect.Value`. All have simplified their generation to cast the signed and unsigned integers to the largest version `int64` and `uint64` to support types with a single generator type. Potentially the narrowing is negligible but for longer runs and large sets of inputs, it might not.

In contrast to the other tools, DropShadow abstracts away the properties expressed in the test cases as well as the process of setting up these cases into the contract itself. This enables an approach where the developer interacts with the test cases through the contracts, which in turn automatically creates the test cases. Additionally, people interested in understanding the function do not have to analyse fragmented test cases but can settle for looking at the contract. This could simplify the test creation process but also enhance maintainability. In Listing 4.7 a contract of the `GetDiscount` function can be seen, which specifies its properties. The automatically generated test cases generated from the contract can be found in Listing B.4.

```go
// region: valid_property
// assume: postalCode = DisjointUnion[int](pkg.LT(1000), pkg.GT(9999))
// assume: month = DisjointUnion[int](pkg.LT(1), pkg.GT(12))
// expect: ret == 0
// region: constant_discount
// assume: postalCode = Inclusive[int](1000, 1999)
// assume: month = Inclusive[int](1, 12)
// expect: ret == 20
// region: seasonal_discount
// assume: postalCode = Inclusive[int](3900, 3999)
// assume: month = Inclusive[int](6, 8)
// expect: ret == 40
// region: seasonal_default
// assume: postalCode = Inclusive[int](3900, 3999)
// assume: month = DisjointUnion[int](pkg.Inclusive(1, 5), pkg.Inclusive(9,
    12))
// expect: ret == 10
// region: default_discount
// assume: postalCode = DisjointUnion[int](pkg.Inclusive(2000, 3899),
    pkg.Inclusive(4000, 9999))
// assume: month = Inclusive[int](1, 12)
// expect: ret == 10
func GetDiscount(postalCode int, month int) int {
    ...
}
```

Listing 4.7: Contract for the `GetDiscount` function expressed with DropShadow.

The properties of the `GetDiscount` function are expressed by 5 regions. Notice how some regions use `DisjointUnion` over multiple generators. Essentially, this allows the merge of multiple equivalent partitions, thus greatly reducing the contract size, which otherwise would have several regions to describe a single equivalent partition. The contract size problem can to some degree also be seen as similar to the path explosion problem, since branching code often defines new partitions.

```go
func Test_GetDiscount_seasonal_discount_Property(t *testing.T) {
  postalCodeGenerator := pkg.Inclusive[int](3900, 3999)
  monthGenerator := pkg.Inclusive[int](6, 8)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    postalCode := postalCodeGenerator.NextValue()
    month := monthGenerator.NextValue()
    GetDiscount1(postalCode, month)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
```

Listing 4.8: An automatically generated test case by DropShadow for the `GetDiscount` function which checks the seasonal discount property.

Through the DropShadow pipeline, the functions under test are injected with their contracts containing the properties, which are expressed through run-time assertion checks directly in the function. The `GetDiscount` function which has been instrumented with the contract can be seen in Listing B.5. As such the structure of the DropShadow test case is simple, following the *arrange, act* structure. This can be seen in Listing 4.8 which shows one of these automatically generated test cases. Essentially, the test cases' only purpose is to invoke the function under test with different data from the generators to evaluate the run-time assertions. This is in contrast to the other tools mentioned, where the test cases are used to express the properties of the functions and contain the checks to see if these are fulfilled.

To further simplify the DropShadow test cases, the configurations of function invocations and time limits can be abstracted away as seen in the other tools. By doing so we can reduce the test cases to simply contain the generators and the function invocation, which is similar to *go/quick*.

4.2.1 Chaining Contracts

DropShadow's approach to instrument contracts directly within the Go source code not only facilitates the checking of individual functions against their specified behaviours but also allows for testing whether the caller functions follow the specification of the callee functions. This inherently gives DropShadow the ability to check for contractual adherence across multiple function calls, which can give the effect of a chain. This effect is a byproduct of using contracts in the manner DropShadow does. To show an example of this consider Listing 4.9.

```
// region: Zero
// assume: value = Constant[float64](0)
// expect: ret == 0
// region: Positive
// assume: value = GT[float64](0)
// expect: ret > 0
func Sqrt(value float64) float64 {
  return math.Sqrt(value)
}

// region: BothPos
// assume: a = Inclusive[float64](1, 1000)
// assume: b = Inclusive[float64](1, 1000)
// expect: ret > 0
func Pythagoras(a, b float64) float64 {
  return Sqrt(a*a + b*b)
}
```

Listing 4.9: Example of caller (Pythagoras) and callee (Sqrt) functions with DropShadow contracts.

Here we see the caller function `Pythagoras` and the callee function `Sqrt`. In the contract for `Sqrt` it is evident, that only an input value equal to or greater than zero is accepted. This makes good sense since taking the square root of a non-zero number is not well-defined in real analysis. In case these expectations are met, the contract promises to return a value equal to or greater than zero, respectively. In `Pythagoras`' contract, we specify that only positive input values (`a` and `b`) are accepted since it does not make sense to calculate the hypotenuse of a triangle, that has any side with a value that is not greater than zero.

When the DropShadow pipeline is invoked, the functions will be instrumented with their contracts, where run-time assertion checks are used to determine whether they adhere to their contract. Moreover, test cases for both functions are automatically generated and executed. What happens is both the `Pythagoras` and `Sqrt` functions both have their test cases. However, as `Sqrt` is a callee function of `Pythagoras`, the instrumented `Sqrt` function is tested in 2 different settings, its own generated test cases and in the test cases of `Pythagoras`. This is beneficial as it allows us to test the `Sqrt` function in isolation, with inputs directly from the generator, where the expected behaviour of the property can be checked. Furthermore, it allows us to check when the function is used as a callee, where the input flows differently from a higher-level function, and as such allows us to check for contract violation use. This example shows a single link in the chain, however one can imagine a more complex program, where more functions are involved.

This chaining approach allows us to be more confident about the high-level behaviour of individual functions than is typically the case with standard, isolated unit testing. In this example, it could be argued, that both the `Zero` and `Positive` regions in `Sqrt`'s contract are superfluous since this function implicitly benefits from `Pythagoras`' contract, which states that only positive input values are allowed. However - there is no guarantee, that `Pythagoras` is the only function calling `Sqrt` - now or in future versions of the program. Therefore, even though some checks might seem unnecessary at the current moment, the thorough and consistent use of contracts can help ensure, that each program component functions correctly both individually and in combination with others. This can act as a safeguard enabling developers to enhance and extend the codebase with more confidence in the future since the contracts will provide immediate feedback on potential violations.

It is worth noting, that the automatic enforcement of pre- and post-conditions when using a tool like DropShadow introduces a spin on the traditional notion of defensive programming and Defense in Depth. The usual way of handling potential errors and abnormal inputs, where the programmer has to handle these cases in the code explicitly, can be somewhat delegated to the contracts since the tool will inject the checks itself. Moreover, using contracts in this way also offers the capability to disable the injected checks in the production environment, if deemed necessary. This flexibility facilitates a more dynamic and adaptable approach to defensive programming accommodating different needs without compromising the integrity of the software.

4.3 Hyperproperties

In this section, we will extend the `Pythagoras` example from the previous section to show, how DropShadow can be used to check for a hyperproperty. `Pythagoras` essentially returns the length of the hypotenuse of a triangle with catheti lengths `a` and `b` (see Listing 4.9). This calculation must be commutative, that is - the hypotenuse must have the same length if the catheti swap their respective lengths (change names, in effect).

Since commutativity is a symmetric relation, we can use the above-mentioned example to show, how DropShadow can be used to check for such a property. Checking a hyperproperty requires comparing the outcomes of two different executions so that the order of operations does not affect the final result. Listing 4.10 shows how the contract for `Pythagoras` has been updated to check for the commutativity property when calculating the hypotenuse of a given, right-angle triangle. The contract and implementation of `Sqrt` remains the same as in Listing 4.9.

```
1  // region: BothPos
2  // assume: a = Inclusive[float64](1, 1000)
3  // assume: b = Inclusive[float64](1, 1000)
4  // expect: ret > 0
5  // hyper:
6  // assume: a_p = b
7  // assume: b_p = a
8  // expect: ret_p == ret
9  func Pythagoras(a, b float64) float64 {
10   return Sqrt(a*a + b*b)
11 }
```

Listing 4.10: A Pythagoras function with a DropShadow contract specifying the commutativity property.

In lines 6-7 within the hyper part of Pytahgoras' contract, we see how a_p gets assigned the value of b and b_p the value of a, whereafter the equality of the two executions' return values gets asserted in line 8. Hence - if this assertion passes, we can conclude that the commutativity property of Pythagoras is upheld. The Pythagoras function which has been instrumented with the contract can be seen in Listing C.1, and the corresponding generated tests in Listing C.2

Listing 4.11 shows another example, where the commutativity property is violated. By a glance, one should expect the property to hold since incrementing a number, and thereby changing it, should create an inequality between the two values. However, due to degrading precision arising from dealing with the very large floating-point numbers seen in lines 1-2, one would find that it does not hold.

```
1  // assume: a = Constant[float64](16777216000000000000)
2  // assume: b = Constant[float64](16777216000000000000)
3  // expect: true
4  // hyper:
5  // assume: a_p = a
6  // assume: b_p = b+1
7  // expect: ret != ret_p
8  func add(a, b float64) float64 {
9    return a + b
10 }
```

Listing 4.11: A function with a DropShadow contract violating the commutativity property.

In other words - when we increment b by 1 in line 6, this small change is not reflected in the value stored in b_p due to precision limits. Consequently - the addition a + b yields the same result as a + (b+1), thus violating the check in line 7. To sum up - the failure to uphold the commutativity property in this example stems from the limitations in the floating point precision of large numbers in computer arithmetic. The add function instrumented with the contract can be seen in Listing C.3, and the corresponding generated tests in Listing C.4

4.3.1 gocontracts

Even though the gocontracts tool also enables design-by-contract in Go, it is worth noting, that if developers using this tool wish to test hyperproperties, such checks must be implemented manually, in contrast to DropShadow where this is implemented automatically by using sequential self-composition. In Listing 4.12 we see how sequential self-composition can be implemented through gocontracts. Notice it is a separate function named PytSelfComp, which invokes the original Pythagoras function. In gocontracts preambles can be used to define code snippets which will be inserted between the pre-conditions and the actual implementation. One might think sequential self-composition could simply be accomplished by invoking the Pythagoras function in the preamble of its contract, similar to DropShadow. However, this recursive call causes an infinite loop. Thus, a separate function is necessary.

```go
// PytSelfComp preamble:
// * a_p := b
// * b_p := a
// * ret_p := Pythagoras(a_p, b_p)
// PytSelfComp ensures:
// * ret == ret_p
func PytSelfComp(a, b float64) float64 {
  a_p := b
  b_p := a
  ret_p := Pythagoras(a_p, b_p)

  ret := Pythagoras(a, b)

  defer func() {
    if !(ret == ret_p) {
      panic("ret == ret_p violated")
    }
  }()
  return ret
}
```

Listing 4.12: Sequential self-composition example using gocontracts for the Pythagoras function.

With this approach, it is not the original the Pythagoras function which has a gocontract defined and is instrumented, but rather the separate function PytSelfComp. Following this approach prevents contracts from being chained, as we will have to define a separate function to test the original function.

Chapter 5

Discussion

In this chapter, we take a critical look at the limitations encountered in our study, thus engaging in a discussion elucidating the implications of these shortcomings. These reflections are aimed at understanding the impact of the limitations of our results and exploring potential strategies for addressing them in future. By analysing the constraints of our methodologies and findings, we aim to provide an understanding of the areas, where enhancements and further investigations could be beneficial.

5.1 `os.Exit(1)`

One of the challenges in DropShadow relates to the information and communication between the test cases and the functions under test once a violation has been found. As the assertions are inserted in the function and not the test cases, obtaining information about a test case execution becomes challenging. Different mechanisms can be employed to signal contract violations, each with its drawbacks. In this section, we will explore different approaches.

Currently DropShadow employs `os.Exit(1)` to signal test failures, and while this method is straightforward, it may not be the best approach for managing test failures. This is due to the immediate and abrupt termination of the test suite. As a result, this prevents the execution of other test cases, which could potentially contain violations. Furthermore, as the context is lost, we have no information about the execution of the function and the generated inputs. Hence we cannot determine whether a violation of a pre-condition, post-condition or hyperproperty has occurred in a given test case.

One alternative is utilising Go's error-handling paradigm. Errors can provide a more controlled way of signalling test failures, allowing for the propagation of detailed and custom error messages. Following this approach, we can create a custom error to indicate a contract violation. This allows the test case to simply check whether the function under test returns such an error. However, relying on errors necessitates, in some cases, altering the signature of the function under test to return an error. For instance, imagine the function: `func Foo() int`. If we rely on Go's error-handling mechanisms to communicate contract violations, it would require an alteration of the function signature to: `func Foo() (int, error)`. This error would have to propagate throughout the whole program, as every use of this function would also have to implement the error, to comply with the updated signature and not cause compilation errors. This might be seen as a cumbersome and intrusive practice and may not align with the original design of the code. Furthermore, this approach might also complicate the test code, requiring additional checks and handling of error values. Additionally, if a function already returns an error, a callee might inadvertently treat a violation error like any other error that the function typically returns. It is a common paradigm in Go to just check if an error is not `nil` and have the same case for all errors.

Using panics is another alternative. They can signal a test failure more dramatically than regular error handling and work like exceptions in other languages. However, they carry the drawback of potentially originating from deeper layers of the code, making it challenging to discern whether the panic was intended for violation signalling or resulted from an actual error in the code.

Probably, the most preferred approach, but maybe also the most difficult one, is in some way to provide a `*testing.T` reference to the function under test. Thereby, the violations can be integrated seamlessly with Go's testing framework, enabling the use of built-in functions like `t.FailNow` and `t.Errorf` to report test failures. This approach maintains the execution flow, allowing multiple test cases to run and report independently, thus preserving the continuity of the test suite and facilitating a comprehensive evaluation of the codebase under test. However, in practice the references must be passed without altering the function signatures and also be thread-safe, accommodating scenarios where multiple tests run concurrently. One could force all testing to be done sequentially and share a global reference. However, we suspect the performance decline would be significant in most development settings and most likely will in a way break some code. Since the focus of the project was to demonstrate the feasibility of including hyperproperties in contracts, we leave this challenge up to future works. It does not disprove hyperproperties' use in contracts but rather is a limitation to the process of testing.

5.2 State

In Section 2.4 we presented sequential self-composition. In it, we described the requirement of states being the same for every subsequent call in the composition. There are multiple ways of looking at the strictness of this requirement. Therefore, in this section, we will present the potential implications relative to DropShadow.

If the various invocations altered the state of the program, and it is not reset, then, in case of a contract breach, we would not know if it was the actual parameters which caused it, or if it was the sequence of calls. In the case of testing, we might not care too much, since we just want to know if there was a breach. So this requirement might not be strictly necessary.

In cases, where the state is required to be reset before a sequential composed call, the solution can be to record the state before the first call and set the state to this value before the second (provided that no pointers are involved as new instances should then be created of the underlying object). However, DropShadow applies sequential self-composition in the function under test, and since the contracts only enable descriptions of input parameters, it is insufficient to describe state resetting. A solution to this would simply be to add a parameter to the function under test describing the state, that must apply. However, this would result in breaking callers to the function, as the signature is required to change.

Extending the contract thus seems like the only approach which can support state resetting. However, as mentioned earlier, we only care for breaches. If it was the sequence of invocations, that caused a breach, then we still found one, which makes us argue, that state resetting is not a strict requirement. But how should we present breaches, that do not limit themselves to involve input/output pairs? Let's say, that all tests (including various runs of the same sequentially self-composed function under test) are run sequentially and they all extend each other. Conceptually, it is the same thing as taking all tests and putting them in sequence in one function. Now, with all tests combined into one, we can store the full trace from start to end as an example of a potential breach. But this composition only considers one permutation, and therefore we would have to alter the sequence of composed tests within it to cover all possible permutations. An optimisation could be to just look at the function which changes the shared state (between tests). However, it can still explode very quickly.

In conclusion, there are several approaches to looking at breaches, depending on whether state reset is used or not. Without reset, we must consider the sequence of test calls, which requires all permutations to be tested. With reset, we can with variable control without considering the previous calls. With all this in mind, DropShadow's main focus lies on introducing contractual design with hyperproperties, and the expressivity they introduce into the code.

5.3 Input Distributions

An important aspect of generating inputs is often what distribution over the inputs should be applied, which in other words is an attempt to tackle how more interesting inputs should be prioritised (those which uncover new or interesting areas of a program, or some which can be more error-prone like boundary values). Often programmers are off by one when indexing an array, or they might have forgotten a division by 0 check or nil pointer checks. DropShadow attempts to uniformly generate inputs within its standard provided generators. These generators, even the numeric ones, do not consider the concept of boundary values.

```
 1 // assume: a = AnyNumber[int]()
 2 // assume: b = AnyNumber[int]()
 3 // expect: ret >= 0
 4 // config: 100000it
 5 func AbsDiv(a, b int) int {
 6   result := a / b
 7   if result < 0 {
 8     result = -result
 9   }
10   return result
11 }
```

Listing 5.1: Division with an absolute result containing two errors related to division by zero and negation of int minimum invalidating its contract.

In Listing 5.1 we have a function `AbsDiv` dividing two numbers and returning the absolute value. The inputs can be any `int` including both extremes, and the result should always be positive.

One of the challenges with using DropShadow contracts is, that they are only as good as the developer's understanding of the program being written. As an example of this, consider Listing 5.1 again. The programmer has set out to write a function, that returns the absolute value of a given integer division, and the contract rightfully reflects, that for any combination of `a` and `b`, the number returned should be 0 or positive.

However, the programmer lacks the insight that there are a few values, that cause `AbsDiv` to behave unintentionally. First of all, if `b` has the value 0, we see the classic division-by-zero error in line 6, in which case `AbsDiv` does not return an integer greater than or equal to zero as the contract promises; instead a `panic` is thrown. Secondly, if `a` has the value `a=INT_MIN` and `b` has the value -1, negating `result` in line 8 will result in an integer overflow, in which case `result` will wrap around to the value `a=INT_MIN` again, causing `AbsDiv` to return a negative value. Like before, this is a breach of the contract.

As can also be seen in line 4 of Listing 5.1, the contract is specified to run the test 1000 times. When considering the range of values an e.g. 64-bit integer can hold, this number of iterations is insufficient to likely reveal the problematic values mentioned earlier. Thus, the programmer should be aware not to fall for any sensation of false security, the use of contracts might wrongfully convey. The specification of contracts does not guarantee any code correctness, and neither will the test cases generated based on these contracts be sure to catch any problematic behaviour.

Listing 5.2 reflects a contract, that takes into account the issues highlighted above. In this new contract, we see how `a` is now assumed to only take on values greater than `INT_MIN` and `b` all values different than 0.

```go
// assume: a = GT[int](math.MinInt)
// assume: b = DisjointUnion[int](pkg.LT(0), pkg.GT(0))
// expect: ret >= 0
// config: 1000it
func AbsDiv(a, b int) int {
  ...
}
```

Listing 5.2: Corrected division with absolute result where the pre-conditions now rejects the previous errors.

The two only apparent approaches which aim to alleviate the above-mentioned problem in the contracts are: first, construct smaller regions focusing on boundary values. The user can, e.g., construct a `DisjointUnion` over the boundaries and then set the test duration to a satisfactory amount. Second, custom generators prioritising the generation of boundary values can be implemented. Considering Listing 5.1, boundary value analysis should consider all combinations of pairs of boundary values.

When DropShadow generates tests, the inputs are generated independently. In other words, the generation of one parameter does not influence the generation of another, and they are not generated as sets of parameters. Thereby, the approach to test generation won't fully support boundary value analysis. A potential solution to this would be a 2-step test. First, the product of interesting inputs from all generators is tested (for those without interesting inputs, a random one is chosen). Second, we randomly test the target. Random testing is still useful as it exercises the developers' assumptions of the implementation in potentially unexpected ways. In the case of Listing 5.1, interesting inputs could be `a=INT_MIN` and `b=1`. Considering the pair of interesting inputs, it can be faster to catch the overflow caused by `-INT_MIN` in line 8.

One way to find the problematic values we have just encountered would be to use a verification technique such as symbolic execution. With this technique, potential code issues within the code can be systematically and automatically explored, without the need for relying on test cases to capture them. Symbolic execution would involve analysing `AbsDiv` by treating inputs as symbolic variables rather than concrete values. This allows for the exploration of the paths, the program can take depending on the conditions applied to these symbolic inputs. By doing so, it can comprehensively evaluate which paths lead to errors like the division-by-zero or integer overflow discussed above.

A constraint solver can then be employed to determine the specific values, that cause the unexpected behaviour - as long as the contracts in some way include the unexpected. In the case with `AbsDiv`, this solver should identify 0 as a problematic value for `b` as well as `a=INT_MIN` and -1 problematic values for `a` and `b` respectively.

Shown is the importance of generating inputs in a way that prioritises certain values. Boundary values might prove very useful, but for DropShadow, it can be difficult to generate such parameters at random, and architectural design changes are required to integrate them sufficiently.

To alleviate this issue DropShadow, allows custom generators where users can implement their distributions. However, even with custom generators and correct definitions of contracts, does not eliminate the problem, that is, contract quality is dependent on how well the developer wrote them. The following will in general cover some aspects of contract quality.

5.4 Quality of Contracts and Performance

The contracts written with DropShadow allow any boolean expression for the ”expect” attribute. However, this does not necessarily mean all expressions are equally insightful. Consider `expect: true`: an expectation of not asserting on any output value would only capture crashes - like fuzzing. Consider a function which should never crash, then this expectation would be useful together with an any generator. However, in many other cases, the contracts in general heavily rely on the degree of domain knowledge, the programmer can express.

DropShadow supports only universal quantification over two traces, provided that the function under test can be sequentially self-composed. This restriction limits the hyperproperties, which can be expressed and thereby tested. More universal quantifiers would be possible by first altering the grammar and then the injection to call the wrapped function more times with new primed variables. Considering the necessity of more traces and alternating quantifiers (hyperproperties as a mixture of quantifiers) how necessary are these to express properties relevant to the user? Our focus has been mostly on observational determinism and non-interference, which do not require alternating quantifiers. Especially from a security standpoint, these two are interesting. But also, monotonicity and symmetrical relations such as the commutative property can be expressed with only two traces. However, the transitivity property for example cannot.

As presented earlier, the current configuration of DropShadow parses Go source files, and then instrument functions with their contracts, after which a test file is generated and executed. When the testing has concluded, the test file is deleted and the source file reverted to its original state. This configuration leads to the question of where the responsibility lies, once the source file is reverted to its original, without the run-time assertion checks. If we consider the *gocontracts* tool, their approach is to keep the run-time assertions in the code, which is also possible in DropShadow by configuring the pipeline. However, this could pose performance issues, as DropShadow instrument plenty of code to check contract adherence. In addition to performance, if the contract simply does not reflect reality by e.g. disallowing some sensor input values, then system failure would occur, potentially rendering the system unusable. This begs the question of contracts in production and development.

Chapter 6

Related work

In this section, we will focus our discussion on related work which covers the same areas and utilises the same techniques as DropShadow and techniques which aim to improve the efficiency of testing properties and hyperproperties. DropShadow, can be seen as a tool combining contractual design of software as well as a trace- and hyperproperty-based testing tool, with the entire focus on finding counterexamples by concrete executions of the program. In addition, we will present some related work concerning the verification techniques.

6.1 Contract-Driven Testing

Testing of programs based on contracts has already been studied, and several tools exist for generating unit tests from these contracts. Contract-driven testing, introduced in [1], explains contracts can be used to derive test cases. As contracts can be checked at run time, makes them suitable for testing. Various approaches utilise contracts for unit test generation, such as Praspel [8] for PHP and JSContest [19] for JavaScript.

Most interestingly is the tool Praspel, which shares many functionalities and techniques with DropShadow. It introduces Design-by-Contract in PHP and is adapted to test generation. They use contracts to instrument the source with runtime assertion checks and generation of test data. Moreover, both DropShadow and Praspel describe multiple explicit behaviours, allowing for more complex expressions of properties. These are known as *Behavioral clause* in Praspel and *region* in DropShadow. However, we differ at two levels. First, Praspel relies on exceptions to communicate contract violations and allow the contract to specify expected exceptions. This is possible as exceptions in PHP can have different types, enabling Praspel to utilise custom exceptions. As discussed earlier, improving contract violation is a future work area for DropShadow. Second, our approach involves parsing files and searching for functions with defined contracts. We then perform the necessary actions to conduct runtime assertion checking and automatically generate concrete test cases using the Go testing library. In contrast, Praspel does not generate actual instances of test cases but instead relies on an algorithm, which takes a set of functions to be tested as a parameter. The algorithm then checks contract violations for each function. Leveraging the testing library provides DropShadow with distinct advantages.

Each of the DropShadow test cases can be executed individually, which also provides immediate feedback on the test results without the need to run the whole test suite. Moreover, we can utilise customised testing strategies.

6.2 Property-Based Testing

QuickCheck [4] is a foundational, property-based testing tool initially developed for the Haskell programming language, emphasising (pseudo) randomised test data generation to validate the properties of a program. It automatically generates test cases attempting to falsify asserted, high-level properties about a function. In comparison, DropShadow also adopts property-based testing but extends QuickCheck's methodology by introducing contract-based testing in the Go programming language. Like DropShadow uses the notion of a `region` to bound the input space in which random input is generated, QuickCheck also offers this possibility by allowing the programmer to implement custom generators to constrain the random values generated. Another notable tool for property-based testing, that draws inspiration from Quickcheck, is Hypothesis for the Python programming language [23]. Like QuickCheck, this tool generates test cases to try and falsify properties of functions using randomised inputs as well as introduce usability enhancing features like generating complex data structures and the ability to ”shrink” failing test cases to simpler forms. In comparison, DropShadow also allows for the testing of complex, custom data types - however, this feature is contingent on the programmer's willingness to code the generator from scratch. Finally, Hypothesis makes a point of being easily extensible, resulting in the existence of third-party extensions, like hypothesis-networkx and hypothesis-bio, for specific research applications.

6.3 Hypertests

Property-based tests are well established and have an arsenal of tool support to back the automatic development of property tests. On the other hand, Hypertests which are tests for finite trace hyperproperties requiring a finite amount of traces to produce a counterexample, do not meet demands in tooling [21].

A framework for developing tests for hyperproperties has been defined in [29]. In it, a coverage metric is defined as the proportion of tested input pairs potentially generating different outputs, computed by analysing the program's Control Flow Graph (CFG). One drawback is the necessity of the code to perform static analysis on the source code. Often, trace property testing tools utilise code coverage, which is a metric describing the proportion of a program that has been executed. In contrast, code coverage can be retrieved by grey-box methods whereas construction of a CFG is whitebox. Also in [29], multiple procedures for various ways of testing hyerproperties are presented. Both, hypertesting, like DropShadow and fuzzing of hyperproperties. The main difference is fuzzing continues based on a ”budget” and success is not determined by coverage but by violations observed before exhausting the budget. DropShadow looks more like a fuzzer because it lacks the coverage metric.

LeakFuzzer, is a hyperfuzzer, that is a fuzzer with the capability of comparing multiple executions against each other. Leakfuzzer, [2], extends AFL++, [11], and enables the fuzzing of the non-interference hyperproperty. LeakFuzzer, has segmented input into a public and private part and assumes the output to always be public. After each execution, it stores the hashes of the secret input and public output and compares them with previous executions sharing the same public input but differing secret inputs. This defers the sequential execution which DropShadow has immediately for the primed version. LeakFuzzer reduces the amount of redundant calls by ensuring that an invocation won't happen twice with the cost of more memory. However, it does not have the detailed capability, like DropShadow, to describe relations between the inputs and outputs.

6.4 Symbolic Execution

The result from running the tests generated by DropShadow, is potential counterexamples for both trace and hyperproperties. Most likely, the tests won't have covered the full specification and thereby won't be a proof. Instead, verification techniques can be employed to provide guarantees.

In [21] the desire for automated generation of tests for hyperproperties is mentioned. DropShadow supports this based on the contracts but relies on concrete execution of the function under test. Symbolic execution alleviates the need for concrete executions by exploring programs and assigning symbolic values to variables. However, large systems can become so complex symbolic execution becomes infeasible in practice. To alleviate this, an approach combining the two called concolic execution can be used. An example is the fuzzer Driller which uses the result of symbolic execution to generate new and interesting inputs [33]. Provided contracts are assumed to be correct for a function: then symbolic execution can be pre-constrained, like Driller, to the pre-conditions from a region with a hyperproperty. Therefore, the focus on proving a single hyperproperty can be limited to the area where the region is handled.

By taking a hyperproperty like we have presented for non-interference, and using sequential self-composition to construct a new program. It would be possible to use symbolic execution to look for assertion violations. However, this requires sufficient tooling for the language in question. In our case, we are focused on the Go language which is still in its infancy. This heavily shows from the lack of verification tools of the Go source code, which to our knowledge for example lacks symbolic execution tools.

Chapter 7

Conclusion

With the development of DropShadow, we have undertaken a challenge to enhance the Go programming environment by embedding contractual design of function specifications. The core objective was to enable developers to specify contractual obligations that would allow the automatic generation and testing of trace properties as well as some hyperproperties. The class of hyperproperties supported by DropShadow enables the testing of common security properties such as non-interference.

We found a lack of contractual design tools through documentation in Go, and to our knowledge none exists in any other language, that directly supports hyperproperties. The primary achievement of DropShadow is to show how contracts can be integrated into Go and be used to express hyperproperties, which are reducible through sequential self-composition. The integration into Go is partly presented as an extension of the grammar definition of Go itself. Everything from the injection of the contracts and automatic test generation is done by DropShadow with the information provided in the contracts. Our implementation of this tool solves several fundamental problems encountered with the extension of hyperproperties, such as the reduction technique. Additionally, the generation of primitive and composite custom types are supported by DropShadow, thus allowing developers to specify generators for all possible types as well as custom generation distributions.

We have explored and shown how the contracts can be used to automatically test several trace properties as well as hyperproperties. We have provided several examples, showing how the region design of contracts are expressed as logical check (which resembles partition testing). Furthermore, we have analysed the tool usage patterns and found a general approach (any generation) to ensure a complete contract accounting for all inputs.

The hyperproperties supported by DropShadow only include the ones, which can be reduced through sequential self-composition, with non-interference being an example. But the question remains, whether all relevant security-related properties or more generally: whether all useful hyperproperties lie within that class. DropShadow can be extended to handle sequential self-composition of more than two traces, but whether there is a necessity to do so is unknown.

The biggest issue we see with DropShadow is the lack of readability in the contracts. The tool should be user-friendly enough such that developers could and would want to use it. With the less readable, yet expressive, contracts this might not always be the case.

We have presented several challenges which are still open, such as handling the presentation of counterexamples, as well as reporting failures. Furthermore, we have also presented an explosion problem relating to the state reset requirement in sequential self-composition. Additional work can be done on more termination criteria. When a breach of contract is found, and state reset is in place, input reduction might be desirable.

We have presented some related work, which all describe these approaches. Furthermore, the use of contracts to produce documentation as well as to improve the speed of symbolic execution by directly substituting invocations of functions with their contracts can be looked into, but to our knowledge, no symbolic execution tool is available for Go. By viewing hyperproperties as a test comprising multiple observations with variable control, we can change our perception of input generation away from a naive approach wanting to test everything to an approach focusing on the interaction between variables.

The insights gained from the development and use of DropShadow underscore the potential for similar methodologies to be adapted and applied in other programming environments. Looking toward the future, the principles and techniques implemented by DropShadow could inspire new tools, that prioritise the testing of security properties.

Bibliography

[1] Bernhard K. Aichernig. Contract-based testing. In Bernhard K. Aichernig and T. S. E. Maibaum, editors, *Formal Methods at the Crossroads. From Panacea to Foundational Support, 10th Anniversary Colloquium of UNU/IIST, the International Institute for Software Technology of The United Nations University, Lisbon, Portugal, March 18-20, 2002, Revised Papers*, volume 2757 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2002.

[2] Daniel Blackwell, Ingolf Becker, and David Clark. Hyperfuzzing: black-box security hypertesting with a grey-box fuzzer. *CoRR*, abs/2308.09081, 2023.

[3] chavacava. dbc4go. `https://github.com/chavacava/dbc4go`. Accessed on: 15/5 2024.

[4] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, page 268–279, New York, NY, USA, 2000. Association for Computing Machinery.

[5] Michael R Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K Micinski, Markus N Rabe, and César Sánchez. Temporal logics for hyperproperties. In *Principles of Security and Trust: Third International Conference, POST 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings 3*, pages 265–284. Springer, 2014.

[6] Michael R. Clarkson and Fred B. Schneider. Hyperproperties. In *2008 21st IEEE Computer Security Foundations Symposium*, pages 51–65, 2008.

[7] Eiffel. Eiffel. `https://www.eiffel.org/`. Accessed on: 16/5 2024.

[8] Ivan Enderlin, Frédéric Dadeau, Alain Giorgetti, and Abdallah Ben Othman. Praspel: A specification language for contract-based testing in PHP. In Burkhart Wolff and Fatiha Zaïdi, editors, *Testing Software and Systems - 23rd IFIP WG 6.1 International Conference, ICTSS 2011, Paris, France, November 7-10, 2011. Proceedings*, volume 7019 of *Lecture Notes in Computer Science*, pages 64–79. Springer, 2011.

[9] Elahe Fazeldehkordi. Linear temporal logic for hyperproperties (hyperltl). Lecture slides for the course Specification and Verification of Parallel Systems `https://www.uio.no/studier/emner/matnat/ifi/IN5110/h19/slides-xtra/hyperltl.pdf`, 11 2019.

[10] Bernd Finkbeiner. Model checking algorithms for hyperproperties. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, pages 3–16. Springer, 2021.

[11] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. AFL++ : Combining incremental steps of fuzzing research. In Yuval Yarom and Sarah Zennou, editors, *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*. USENIX Association, 2020.

[12] Pact Foundation. Pact go. `https://github.com/pact-foundation/pact-go`. Accessed on: 15/5 2024.

[13] Google. Go doc comments. `https://go.dev/doc/comment`. Accessed on: 24/4 2024.

[14] Google. Go fuzzing. `https://go.dev/doc/security/fuzz/`. Accessed on: 17/5 2024.

[15] Google. The go programming language specification. `https://go.dev/ref/spec`. Accessed on: 10/4 2024.

[16] Google. quick. `https://pkg.go.dev/testing/quick`. Accessed on: 22/5 2024.

[17] Mike Gordon. Linear temporal logic (ltl). Lecture slides for the course Temporal Logic and Model Checking `https://www.cl.cam.ac.uk/archive/mjcg/TempLogic/Lectures/L4.Jan27.pdf`, 2015.

[18] Thomas Hamilton. 7 principles of software testing with examples. `https://www.guru99.com/software-testing-seven-principles.html`. Accessed on: 3/5 2024.

[19] Phillip Heidegger and Peter Thiemann. Jscontest: Contract-driven testing and path effect inference for javascript. *J. Object Technol.*, 11(1):1–29, 2012.

[20] Shayan Hosseini. grpc-go-contracts. `https://github.com/shayanh/grpc-go-contracts`. Accessed on: 15/5 2024.

[21] Johannes Kinder. Hypertesting: The case for automated testing of hyperproperties. In *3rd Workshop on Hot Issues in Security Principles and Trust (HotSpot)*, pages 1–8, 2015.

[22] leanovate. gopter. `https://github.com/leanovate/gopter`. Accessed on: 22/5 2024.

[23] David Maciver and Zac Hatfield-Dodds. Hypothesis: A new approach to property-based testing. *J. Open Source Softw.*, 4(43):1891, 2019.

[24] Brian A. Malloy and Jeffrey M. Voas. Programming with assertions: A prospectus. *IT Prof.*, 6(5):53–59, 2004.

[25] Bertrand Meyer. *Introduction to the Theory of Programming Languages*. Prentice-Hall, 1990.

[26] Mitre. Cve-2014-0160. `https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-0160`. Accessed on: 17/5 2024.

[27] OSS-Fuzz. Oss-fuzz. `https://google.github.io/oss-fuzz/`. Accessed on: 17/5 2024.

[28] Parquery. gocontracts. `https://github.com/Parquery/gocontracts`. Accessed on: 15/5 2024.

[29] Michele Pasqua, Mariano Ceccato, and Paolo Tonella. Hypertesting of programs: Theoretical foundation and automated test generation. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, 2024.

[30] Gregory Petrosyan. rapid. `https://github.com/flyingmutant/rapid`. Accessed on: 22/5 2024.

[31] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 46–57. IEEE Computer Society, 1977.

[32] David S. Rosenblum. Towards a method of programming with assertions. In Tony Montgomery, Lori A. Clarke, and Carlo Ghezzi, editors, *Proceedings of the 14th International Conference on Software Engineering, Melbourne, Australia, May 11-15, 1992*, pages 92–104. ACM Press, 1992.

[33] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016.

[34] Synopsys. The heartbleed bug. `https://heartbleed.com/`. Accessed on: 16/5 2024.

Appendix A

Evaluation: Absolute

```go
func Absolute(value float64) float64 {
  values := pkg.Constant[float64](0)
  valuess := pkg.LT[float64](0)
  valuesss := pkg.GT[float64](0)
  valuessss := pkg.AnyNumber[float64]()
  fRegion := func(value float64) []int {
    ret := make([]int, 0)
    if values.Contains(value) {
      ret = append(ret, 0)
    }
    if valuess.Contains(value) {
      ret = append(ret, 1)
    }
    if valuesss.Contains(value) {
      ret = append(ret, 2)
    }
    if valuessss.Contains(value) {
      ret = append(ret, 3)
    }
    return ret
  }
  regions := fRegion(value)
  if len(regions) == 0 {
    os.Exit(1)
  }
  wrap := func(value float64) float64 {
    if value < 0 {
      return -value
    }
    return value
  }
  ret := wrap(value)
  for _, region := range regions {
    switch region {
    case 0:
      if !(ret == 0) {
        os.Exit(1)
      }
    case 1:
      if !(ret == -value) {
        os.Exit(1)
      }
    case 2:
      if !(ret == value) {
        os.Exit(1)
      }
    case 3:
      if !(ret >= 0) {
        os.Exit(1)
      }
    }
  }
  return ret
}
```

Listing A.1: The Absolute function with DropShadow injected contracts.

```
func Test_Absolute_Zero_Property(t *testing.T) {
  valueGenerator := pkg.Constant[float64](0)
  iterations, iterationLimit := 0, 1
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    value := valueGenerator.NextValue()
    Absolute(value)
    iterations++
    if iterations >= iterationLimit {
      return
    }
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_Absolute_Negative_values_Property(t *testing.T) {
  valueGenerator := pkg.LT[float64](0)
```

```go
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    value := valueGenerator.NextValue()
    Absolute(value)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_Absolute_Positive_values_Property(t *testing.T) {
  valueGenerator := pkg.GT[float64](0)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    value := valueGenerator.NextValue()
    Absolute(value)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_Absolute_All_Property(t *testing.T) {
  valueGenerator := pkg.AnyNumber[float64]()
  iterations, iterationLimit := 0, 100
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    value := valueGenerator.NextValue()
    Absolute(value)
    iterations++
    if iterations >= iterationLimit {
      return
    }
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
```

Listing A.2: Test cases that has been automatically generated by DropShadow for the Absolute function.

```go
func Absolute(value float64) (result float64) {
  // Post-condition
  defer func() {
    if !(value < 0 && result == -value || value >= 0 && result == value) {
      panic("Violated: value < 0 && result == -value || value >= 0 && result
          == value")
    }
  }()

  result = math.Abs(value)
  return
}
```

Listing A.3: The Aboslute function with contracts injected by the gocontracts tool.

Appendix B

Evaluation: GetDiscount

```go
func TestValidityPropertyRapid(t *testing.T) {
  rapid.Check(t, func(t *rapid.T) {
    postalCode := rapid.Int().Draw(t, "postalCode")
    month := rapid.Int().Draw(t, "month")
    isValid := postalCode >= 1000 && postalCode <= 9999 && month >= 1 && month
        <= 12

    result := GetDiscountOtherTools(postalCode, month)

    if !isValid && result != 0 || isValid && result == 0 {
      t.Fatalf("Expected 0 for invalid inputs")
    }
  })
}

func TestConstantDiscountPropertyRapid(t *testing.T) {
  rapid.Check(t, func(t *rapid.T) {
    postalCode := rapid.IntRange(1000, 1999).Draw(t, "postalCode")
    month := rapid.IntRange(1, 12).Draw(t, "month")

    result := GetDiscountOtherTools(postalCode, month)

    if result != 20 {
      t.Fatalf("Constant discount property violation")
    }
  })
}

func TestSeasonalDiscountPropertyRapid(t *testing.T) {
  rapid.Check(t, func(t *rapid.T) {
    postalCode := rapid.IntRange(3900, 3999).Draw(t, "postalCode")
    month := rapid.IntRange(1, 12).Draw(t, "month")
    isSummer := month >= 6 && month <= 8

    result := GetDiscountOtherTools(postalCode, month)

    if (isSummer && result != 40) || (!isSummer && result != 10) {
      t.Fatalf("Seasonal discount property violation")
    }
  })
}

func TestDefaultDiscountPropertyRapid(t *testing.T) {
  rapid.Check(t, func(t *rapid.T) {
    postalCode := rapid.IntRange(2000, 9999).Draw(t, "postalCode")
    month := rapid.IntRange(1, 12).Draw(t, "month")

    isValid := (postalCode >= 2000 && postalCode <= 3899) || (postalCode >=
       4000 && postalCode <= 9999) && month >= 1 && month <= 12
    result := GetDiscountOtherTools(postalCode, month)

    if isValid && result != 10 {
      t.Fatalf("Default discount property violation")
    }
  })
}
```

Listing B.1: Manually written property-based test cases using the rapid tool for the GetDiscount function.

```go
func TestValidityPropertyGopter(t *testing.T) {
  properties := gopter.NewProperties(nil)

  properties.Property("Validity property", prop.ForAll(
    func(postalCode int, month int) bool {
      result := GetDiscountOtherTools(postalCode, month)
      isValid := postalCode >= 1000 && postalCode <= 9999 && month >= 1 &&
         month <= 12
      if !isValid {
        return result == 0
      }
      return result != 0
    },
    gen.Int(),
    gen.Int(),
  ))

  properties.TestingRun(t)
}

func TestSeasonalDiscountPropertyGopter(t *testing.T) {
  properties := gopter.NewProperties(nil)

  properties.Property("Seasonal discount property", prop.ForAll(
    func(postalCode int, month int) bool {
      result := GetDiscountOtherTools(postalCode, month)
      isSummer := month >= 6 && month <= 8
      if postalCode >= 3900 && postalCode <= 3999 {
        if isSummer {
          return result == 40
        }
        return result == 10
      }
      return true
    },
    gen.IntRange(3900, 3999),
    gen.IntRange(1, 12),
  ))

  properties.TestingRun(t)
}

func TestConstantDiscountPropertyGopter(t *testing.T) {
  properties := gopter.NewProperties(nil)

  properties.Property("Constant discount property", prop.ForAll(
    func(postalCode int, month int) bool {
      result := GetDiscountOtherTools(postalCode, month)
      return postalCode >= 1000 && postalCode <= 1999 && result == 20
    },
    gen.IntRange(1000, 1999),
    gen.IntRange(1, 12),
  ))

  properties.TestingRun(t)
}

func TestDefaultPropertyGopter(t *testing.T) {
  properties := gopter.NewProperties(nil)

  properties.Property("Default discount property", prop.ForAll(
    func(postalCode int, month int) bool {
      result := GetDiscountOtherTools(postalCode, month)
      isValid := (postalCode >= 2000 && postalCode <= 3899) || (postalCode >=
          4000 && postalCode <= 9999) && month >= 1 && month <= 12
      if isValid {
        return result == 10
      }
      return true
    },
    gen.IntRange(2000, 9999),
    gen.IntRange(1, 12),
  ))

  properties.TestingRun(t)
}
```

Listing B.2: Manually written property-based test cases using the gopter tool for the GetDiscount function.

```go
func TestValidityPropertyQuick(t *testing.T) {
  property := func(month, postalCode int) bool {
    result := GetDiscountOtherTools(postalCode, month)

    isValid := postalCode >= 1000 && postalCode <= 9999 && month >= 1 && month
        <= 12

    if !isValid {
      return result == 0
    }
    return result != 0
  }

  config := &quick.Config{ MaxCount: 1000 }

  if err := quick.Check(property, config); err != nil {
    t.Error("Failed validity test:", err)
  }
}

func TestSeasonalDiscountPropertyQuick(t *testing.T) {
  property := func(month, postalCode int) bool {
    discount := GetDiscountOtherTools(postalCode, month)
    if month >= 6 && month <= 8 {
      return discount == 40 // Summer discount
    }
    return discount == 10
  }

  config := &quick.Config{
    MaxCount: 1000,
    Values: func(v []reflect.Value, r *rand.Rand) {
      v[0] = reflect.ValueOf(1 + r.Intn(12)) // Month: [1; 12]
      v[1] = reflect.ValueOf(3900 + r.Intn(100)) // Postal code: [3900; 3999]
    },
  }

  if err := quick.Check(property, config); err != nil {
    t.Error("Failed seasonal discount test:", err)
  }
}

func TestConstantDiscountPropertyQuick(t *testing.T) {
  property := func(month, postalCode int) bool {
    if postalCode >= 1000 && postalCode <= 1999 {
      return GetDiscountOtherTools(postalCode, month) == 20
    }
    return true
  }

  config := &quick.Config{
    MaxCount: 1000,
    Values: func(v []reflect.Value, r *rand.Rand) {
      v[0] = reflect.ValueOf(1 + r.Intn(12)) // Month: [1; 12]
      v[1] = reflect.ValueOf(1000 + r.Intn(1000)) // Postal code: [1000; 1999]
    },
  }

  if err := quick.Check(property, config); err != nil {
    t.Error("Failed constant discount test:", err)
  }
}

func TestDefaultPropertyQuick(t *testing.T) {
  property := func(month, postalCode int) bool {
    if (postalCode >= 2000 && postalCode <= 3899) || (postalCode >= 4000 &&
       postalCode <= 9999) && month >= 1 && month <= 12 {
      return GetDiscountOtherTools(postalCode, month) == 10
    }
    return true
  }

  config := &quick.Config{
    MaxCount: 1000,
    Values: func(v []reflect.Value, r *rand.Rand) {
      v[0] = reflect.ValueOf(1 + r.Intn(12)) // Month: [1; 12]
      v[1] = reflect.ValueOf(1000 + r.Intn(1000)) // Postal code: [2000; 9999]
    },
  }

  if err := quick.Check(property, config); err != nil {
    t.Error("Failed default discount test:", err)
  }
}
```

Listing B.3: Manually written property-based test cases using the go/quick tool for the GetDiscount function.

```
func Test_GetDiscount_valid_property_Property(t *testing.T) {
  postalCodeGenerator := pkg.DisjointUnion[int](pkg.LT[int](1000),
      pkg.GT[int](9999))
  monthGenerator := pkg.DisjointUnion[int](pkg.LT[int](1), pkg.GT[int](12))
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    postalCode := postalCodeGenerator.NextValue()
    month := monthGenerator.NextValue()
    GetDiscount(postalCode, month)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_GetDiscount_constant_discount_Property(t *testing.T) {
  postalCodeGenerator := pkg.Inclusive[int](1000, 1999)
  monthGenerator := pkg.Inclusive[int](1, 12)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    postalCode := postalCodeGenerator.NextValue()
    month := monthGenerator.NextValue()
    GetDiscount(postalCode, month)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_GetDiscount_seasonal_discount_Property(t *testing.T) {
  postalCodeGenerator := pkg.Inclusive[int](3900, 3999)
  monthGenerator := pkg.Inclusive[int](6, 8)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    postalCode := postalCodeGenerator.NextValue()
    month := monthGenerator.NextValue()
    GetDiscount(postalCode, month)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_GetDiscount_seasonal_default_Property(t *testing.T) {
  postalCodeGenerator := pkg.Inclusive[int](3900, 3999)
  monthGenerator := pkg.DisjointUnion[int](pkg.Inclusive[int](1, 5),
      pkg.Inclusive[int](9, 12))
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    postalCode := postalCodeGenerator.NextValue()
    month := monthGenerator.NextValue()
    GetDiscount(postalCode, month)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_GetDiscount_default_discount_Property(t *testing.T) {
  postalCodeGenerator := pkg.DisjointUnion[int](pkg.Inclusive[int](2000,
      3899), pkg.Inclusive[int](4000, 9999))
  monthGenerator := pkg.Inclusive[int](1, 12)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    postalCode := postalCodeGenerator.NextValue()
    month := monthGenerator.NextValue()
    GetDiscount(postalCode, month)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
```

Listing B.4: The automatically generated test cases by DropShadow for the GetDiscount function.

```
func GetDiscount(postalCode int, month int) int {
  postalCodes := pkg.DisjointUnion[int](pkg.LT[int](1000), pkg.GT[int](9999))
  months := pkg.DisjointUnion[int](pkg.LT[int](1), pkg.GT[int](12))
  postalCodess := pkg.Inclusive[int](1000, 1999)
  monthss := pkg.Inclusive[int](1, 12)
  postalCodesss := pkg.Inclusive[int](3900, 3999)
  monthsss := pkg.Inclusive[int](6, 8)
  postalCodessss := pkg.Inclusive[int](3900, 3999)
  monthssss := pkg.DisjointUnion[int](pkg.Inclusive[int](1, 5),
      pkg.Inclusive[int](9, 12))
  postalCodesssss := pkg.DisjointUnion[int](pkg.Inclusive[int](2000, 3899),
      pkg.Inclusive[int](4000, 9999))
  monthsssss := pkg.Inclusive[int](1, 12)
  fRegion := func(postalCode int, month int) []int {
    ret := make([]int, 0)
    if postalCodes.Contains(postalCode) && months.Contains(month) {
      ret = append(ret, 0)
    }
    if postalCodess.Contains(postalCode) && monthss.Contains(month) {
      ret = append(ret, 1)
    }
    if postalCodesss.Contains(postalCode) && monthsss.Contains(month) {
      ret = append(ret, 2)
    }
    if postalCodessss.Contains(postalCode) && monthssss.Contains(month) {
      ret = append(ret, 3)
    }
    if postalCodesssss.Contains(postalCode) && monthsssss.Contains(month) {
      ret = append(ret, 4)
    }
    return ret
  }
  regions := fRegion(postalCode, month)
  if len(regions) == 0 {
    os.Exit(1)
  }
```

```
wrap := func(postalCode int, month int) int {
  if postalCode < 1000 || postalCode > 9999 || month < 1 || month > 12 {
    return 0
  }
  switch {
  case postalCode >= 1000 && postalCode <= 1999:
    return 20
  case postalCode >= 2000 && postalCode <= 3899:
    return 10
  case postalCode >= 3900 && postalCode <= 3999:
    if month >= 6 && month <= 8 {
      return 40
    }
    return 10
  case postalCode >= 4000 && postalCode <= 9999:
    return 10
  default:
    return 0
  }
}
ret := wrap(postalCode, month)
for _, region := range regions {
  switch region {
  case 0:
    if !(ret == 0) {
      os.Exit(1)
    }
  case 1:
    if !(ret == 20) {
      os.Exit(1)
    }
  case 2:
    if !(ret == 40) {
      os.Exit(1)
    }
  case 3:
    if !(ret == 10) {
      os.Exit(1)
    }
  case 4:
    if !(ret == 10) {
      os.Exit(1)
    }
    }
  }
  return ret
}
```

Listing B.5: The GetDiscount function which has been instrumented with the DropShadow contract.

Appendix C

Evaluation: Hyperproperties

```
func Pythagoras(a, b float64) float64 {
  as := pkg.Inclusive[float64](1, 1000)
  bs := pkg.Inclusive[float64](1, 1000)
  fRegion := func(a, b float64) []int {
    ret := make([]int, 0)
    if as.Contains(a) && bs.Contains(b) {
      ret = append(ret, 0)
    }
    return ret
  }
  regions := fRegion(a, b)
  if len(regions) == 0 {
    os.Exit(1)
  }
  wrap := func(a, b float64) float64 {
    return Sqrt(a*a + b*b)
  }
  ret := wrap(a, b)
  for _, region := range regions {
    switch region {
    case 0:
      if !(ret > 0) {
        os.Exit(1)
      }
    }
  }
  for _, region := range regions {
    switch region {
    case 0:
      a_p := b
      b_p := a
      ret_p := wrap(a_p, b_p)
      if !(ret_p == ret) {
        os.Exit(1)
      }
    }
  }
  return ret
}
```

Listing C.1: DropShadow injected contract for Pythagoras.

```
func Test_Sqrt_Zero_Property(t *testing.T) {
  valueGenerator := pkg.Constant[float64](0)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    value := valueGenerator.NextValue()
    Sqrt(value)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_Sqrt_Positive_Property(t *testing.T) {
  valueGenerator := pkg.GT[float64](0)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    value := valueGenerator.NextValue()
    Sqrt(value)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
func Test_Pythagoras_BothPos_Property(t *testing.T) {
  aGenerator := pkg.Inclusive[float64](1, 1000)
  bGenerator := pkg.Inclusive[float64](1, 1000)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    a := aGenerator.NextValue()
    b := bGenerator.NextValue()
    Pythagoras(a, b)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
```

Listing C.2: The automatically generated test cases by DropShadow for the Sqrt and Pythagoras functions.

```
func add(a, b float64) float64 {
  as := pkg.Constant[float64](16777216000000000000)
  bs := pkg.Constant[float64](16777216000000000000)
  fRegion := func(a, b float64) []int {
    ret := make([]int, 0)
    if as.Contains(a) && bs.Contains(b) {
      ret = append(ret, 0)
    }
    return ret
  }
  regions := fRegion(a, b)
  if len(regions) == 0 {
    os.Exit(1)
  }
  wrap := func(a, b float64) float64 {
    return a + b
  }
  ret := wrap(a, b)
  for _, region := range regions {
    switch region {
    case 0:
      if !(true) {
        os.Exit(1)
      }
    }
  }
  for _, region := range regions {
    switch region {
    case 0:
      a_p := a
      b_p := b + 1
      ret_p := wrap(a_p, b_p)
      if !(ret != ret_p) {
        os.Exit(1)
      }
    }
  }
  return ret
}
```

Listing C.3: The Add function instrumented with its contract.

```
func Test_add_Property(t *testing.T) {
  aGenerator := pkg.Constant[float64](16777216000000000000)
  bGenerator := pkg.Constant[float64](16777216000000000000)
  startTime, timeLimit := time.Now(), 10*time.Second
  for {
    a := aGenerator.NextValue()
    b := bGenerator.NextValue()
    add(a, b)
    if time.Since(startTime) >= timeLimit {
      return
    }
  }
}
```

Listing C.4: The Add function instrumented with its DropShadow contract.