AI-Assisted Programming with Test-based Refinement*

Bernhard K. Aichernig[1] and Klaus Havelund[2]**

[1] Institute of Software Technology, Graz University of Technology, Austria
[2] Jet Propulsion Laboratory, California Inst. of Technology, USA

Abstract. This work explores the utilization of a Large Language Model (LLM), specifically OpenAI's ChatGPT, to develop a program as a sequence of refinements. Traditionally in formal methods literature such refinements are proven correct, which can be time consuming. In this work the refinements are tested using property-based testing. This approach addresses the problem of ensuring that the code generated by an LLM is correct, which is one of the main challenges of code generation with LLMs. Programs are developed in Scala and testing is performed with ScalaCheck. This approach is demonstrated through the development and testing of a classical bridge controller, originally presented in documentation for the refinement-based Event-B theorem prover.

1 Introduction

Neural program synthesis, based on Large Language Models (LLMs) which are trained on open source code, are quickly becoming a popular addition to the software developer's toolbox. Services like, for instance, Open AI's ChatGPT[1], Google's Gemini[2] (the next generation of Bard), and GitHub's Copilot[3], can generate code in many different programming languages from natural language requirements entered as "prompts". A system like ChatGPT is for example particular good at answering questions like *"how do you do X in programming language Y?"*.

Prompt-based programming, however, seems to work best for development of smaller programs. It currently appears infeasible to generate large and complex programs from natural language prompts. It is also not clear how good such a system is for generating code that strays away from common patterns. We refer to this as the *complexity problem*. Furthermore, neural systems do not come with guarantees of producing correct, safe, or secure code, what we shall refer to as the *verification problem*.

** The research performed by this author was carried out at Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration.

[1] https://chat.openai.com

[2] https://gemini.google.com

[3] https://github.com/features/copilot

We propose *test-based refinement* to address these two problems. Program refinement is an old idea, but usually performed with *proofs* of correctness [6, 1, 16, 13, 2]. We experiment with this concept but using *testing*. Specifically, to approach the *complexity* problem, we propose a method based on program refinement, where a program is developed in a step-wise manner, starting with a very high-level abstract program, and then refining it iteratively, towards a final implementation. At each refinement step, the LLM can provide assistance by generating code suggestions and refining existing code snippets. To approach the program *verification* problem, we suggest to apply automated *testing*, including test-case generation, to test that each refinement implements the previous step. Literature on program refinement usually approaches the verification problem as a deductive proof problem. Proofs are, however, hard to carry out for humans, even with automated proof tools.

This work is in line with the TriCo (Triple Co-Piloting) principle to software development described in [3], which argues that implementation, (formal) specification, and tests should be developed hand-in-hand, assisted by machine learning, such as e.g. an LLM. We do, however, not address the formal specification part of TriCo. In general, there has been published numerous articles on the use of LLMs for software development. The paper [7] provides an overview of some of these, in addition to reflecting on how LLMs can be used for software development and quality assurance. Closest to our work is the contribution in [9]. The authors first instantiate computer adventure games with the help of ChatGPT from natural language requirements. Then, they mine finite-state models from the code and check their correctness with model checkers. A yet different approach is to verify neural networks themselves, as discussed in [8].

Our experiments are carried out by developing a classical bridge controller, originally introduced as part of the tutorial for the Event-B theorem prover [2], where a proof-based refinement is performed. Our programs are written in the Scala[4] programming language. Testing is performed using the ScalaCheck[5] property-based testing library. Property-based testing is a method for defining general properties the code should satisfy, quantifying over the input domains (similar to universal quantification in mathematics), and the ScalaCheck library automatically generates test data to verify those properties across a wide range of inputs. Property-based testing is also mentioned in [3] as a way of ensuring correctness in the TriCo approach. We use property-based testing in two ways. First we use property-based testing as a way to test each single program, following the original intent of ScalaCheck. Second, we use property-based testing to test the refinement relation between two versions of a program.

We demonstrate the approach with two different refinements of the bridge controller. The first refinement follows the Event-B proof of the bridge controller, which describes a system as a set of events, each consisting of a precondition and an action having side effects on a shared state. Conceptually one outer loop repeatedly picks an event where the precondition is true and executes the action,

[4] https://www.scala-lang.org

[5] https://scalacheck.org

until no event preconditions are true. This concept has its roots in Dijkstra's language of guarded commands [11], which is Turing complete. The second refinement follows a more classical object-oriented approach, the way many would normally write programs in a language like Scala.

The objective of this work is to (i) explore writing a program iteratively in steps using program refinement, and test the refinements; (ii) augment refinement tests with property-based testing; and (iii) use ChatGPT to generate code, refinement tests and property-based tests. All code should be (and was largely) generated by ChatGPT. Note that we are using ChatGPT's web-based interface Version 4 (and in some cases Version 3 when the upper limit per day for use of Version 4 was reached). We did not use any APIs to the LLM. We also did not use Copilot or any such IDE.

The paper is structured as follows. Section 2 introduces the background for this work, including the well established theory of refinement, in particular its proof theoretic focus, as well as property-based testing. Section 3 explains how one can test refinements instead of proving them correct. Section 4 introduces the bridge controller example. Section 5 presents the guarded command approach to test-based refinement in Scala, while Section 6 presents the object-oriented approach. Finally, Section 7 concludes the paper with observations.

2 Background

In this section we introduce the background for the work, including the established notions of *refinement* and *property-based testing.*

2.1 Refinement

Refinement is the idea of developing a system as a series of specifications (which can be programs), each refining the previous specification, except the first, which constitutes the top-level specification of the system. An implementation at the bottom is then defined as a refinement of the top level abstract specification through transitivity of the refinement relation. A system can be a physical system, a software system, or a mix of the two.

Traditionally in formal methods literature, refinements are proved correct using refinement mappings. In this section we formalize this concept. The theory presented is a minor modification of the theory developed by Abadi and Lamport [1] with the addition of observation functions and invariants as described in [12] (the definitions below are from [12] with a few modifications). We introduce the concepts of *transition systems*, *execution traces*, *invariants*, *refinements*, and *refinement mappings*. Specifications are written as transition systems. A *transition system* is defined as follows.

Definition 1 (Transition System). *A transition system is a five-tuple* $(\Sigma, I, N, \Sigma_o, \pi)$ *where*

- Σ *is a* state *space.*

- $I \subseteq \Sigma$ *is the set of* initial *states.*
- $N \subseteq \Sigma \times \Sigma$ *is the* next-state *relation. Elements of N are denoted by pairs of the form (s, t), meaning that there is a transition from the state s to the state t.*
- Σ_o *is the state space of observations.*
- $\pi : \Sigma \to \Sigma_o$ *is an* observation function*.*

The observation function π in the above definition, which when applied to a state in Σ returns an observation in Σ_o, allows us to compare traces from two transition systems using a shared state space Σ_o in the case where the internal states Σ of the two systems differ.

An *execution trace* is an infinite sequence $\sigma = \langle s_0, s_1, s_2, \ldots \rangle$ of states[6], where the first state satisfies the I predicate and every pair of adjacent states is related by the N relation. We let σ_i denote the i'th element s_i of the sequence. The traces of a transition system can be defined as follows.

Definition 2 (Traces). *The traces of a transition system are defined as follows:*

$$\Theta(\Sigma, I, N, \Sigma_o, \pi) = \{\sigma \in \Sigma^\omega \mid \sigma_0 \in I \wedge \forall i \geq 0 \cdot N(\sigma_i, \sigma_{i+1})\}$$

A projection function π applied to a trace $\langle s_1, s_2, \ldots \rangle$ results in the projected trace $\langle \pi(s_1), \pi(s_2), \ldots \rangle$.

An *invariant* is a state predicate true on all states reachable from an initial state via the next-state relation.

Definition 3 (Invariant). *Given a transition system $S = (\Sigma, I, N, \Sigma_o, \pi)$, then a predicate $P : \Sigma \to \mathcal{B}$ is an S invariant iff*

$$\forall \sigma \in \Theta(S) \cdot \forall i \geq 0 \cdot P(\sigma_i)$$

We can now define the concept of *refinement.*

Definition 4 (Refinement). *A transition system $S_2 = (\Sigma_2, I_2, N_2, \Sigma_o, \pi_2)$* refines *a transition system $S_1 = (\Sigma_1, I_1, N_1, \Sigma_o, \pi_1)$ iff for every trace of S_2 there exists a trace of S_1 with the same observed states:*

$$\forall \sigma_2 \in \Theta(S_2) \cdot \exists \sigma_1 \in \Theta(S_1) \cdot \pi_1(\sigma_1) = \pi_2(\sigma_2)$$

Note that it is here assumed that the abstract transition system S_1 can perform *stuttering* (no-op) steps where the state does not change, reflecting that the concrete system S_2 makes an internal move not corresponding to a state changing move in S_1.

This definition of refinement in terms of infinite traces, however, is not useful for practical proofs or tests. For practical purposes, the notion of a *refinement mapping* from a lower level transition system S_2 to a higher-level one S_1 is introduced, allowing to reason about pairs of states.

[6] A finite execution can be thought of as being represented by an infinite trace with the last state of the finite execution being repeated infinitely.

Definition 5 (Refinement Mapping). *A* refinement mapping *from a transition system $S_2 = (\Sigma_2, I_2, N_2, \Sigma_o, \pi_2)$ to a transition system $S_1 = (\Sigma_1, I_1, N_1, \Sigma_o, \pi_1)$ is a mapping $f : \Sigma_2 \rightarrow \Sigma_1$ such that there exists an S_2 invariant P (representing reachable states in S_2), where:*

1. $\forall s \in \Sigma_2 \cdot \pi_1(f(s)) = \pi_2(s)$
2. $\forall s \in \Sigma_2 \cdot I_2(s) \Rightarrow I_1(f(s))$
3. $\forall s, t \in \Sigma_2 \cdot P(s) \wedge N_2(s,t) \Rightarrow N_1(f(s), f(t))$

The main property is 3, expressing that a move on the concrete transition system S_2 simulates a corresponding move in the abstract transition system S_1, see Figure 1. We can now state the main theorem (which is stated in [1]):

Theorem 1 (Existence of Refinement Mappings). *If there exists a refinement mapping from a transition system S_2 to a transition system S_1, then S_2 refines S_1.*

Defining the refinement mapping turns out typically to be easy, whereas proving that it is indeed a refinement mapping (Property 3 in Definition 5) can be hard, in particular discovering a suffiently strong invariant. As we shall see in Section 3 we will test this relationship instead of proving it.

$$\begin{tikzcd}
f(s) \arrow[r, "N_1"] & f(t) \\
s \arrow[u, "f"] \arrow[r, "N_2"'] & t \arrow[u, "f"']
\end{tikzcd}$$

Fig. 1. Commuting refinement diagram.

2.2 Property-based Testing

Property-based testing (PBT) is a testing technique that tries to falsify a given property by generating random input data and verifying the expected behaviour. The motivation is to replace hand-written unit tests with property specifications that are tested automatically. The first tool implementing this style of testing was QuickCheck [10]. Properties can range from simple algebraic equations to complex state machine models. Like in all model-based testing techniques the properties serve as a source for test-case generation as well as test oracles. It is a well-known testing practice in functional programming, but nowadays we see a growth of applications outside its traditional domain, including automotive software [5] and web-services [15, 4].

A simple example of an algebraic property is that the reverse of the reverse of a list must equal the original list:

$$\forall \; xs \in List[T] : reverse(reverse(xs)) = xs$$

A PBT tool will generate a series of random lists xs, execute the reverse function and evaluate the property. This test-case generation is realized via the composition of test-case generators. For the example above, the test-case generator for lists will use a test-case generator for type T. In addition to default generators for basic types, a user can define custom generators and mix them arbitrarily.

If a property is violated, a counter-example will be presented to the user. For easier debugging, the counter-example is simplified in a process called *shrinking*.

The original QuickCheck tool was implemented in, and supported testing of, Haskell programs, but has been ported to many programming languages including FsCheck[7] for C#, Hypothesis[8] for Python, and ScalaCheck[9] for Scala. In this work we rely on the latter. In ScalaCheck the above property for integer lists would be represented as follows:

```
property("reverse") = forAll { (xs: List[Int]) => reverse(reverse(xs)) == xs}
```

ScalaCheck will by default generate 100 random integer lists in order to test this property.

3 Refinement and Testing

In this section, we present our AI-assisted method for developing programs with LLMs. We will use step-wise refinement in order to develop a program iteratively from an abstract method down to a concrete implementation. Hence, the LLM shall assist us in generating improved versions of an abstract program that preserve the refinement mapping. In each refinement step, additional requirements are considered while preserving correctness with respect to the more abstract versions. Traditionally, the refinement relation is formally verified via model checking, like in TLA [14], or via theorem proving, like in Event-B [2].

In our method, we propose a more lightweight approach, namely the automated testing of the refinement relation with the help of property-based testing. More concretely, we use a refinement property and data generators to test if two objects respect refinement.

For the testing of refinement, we can simplify the definition of refinement in Definition 5. Assuming executable Scala methods to be tested, we can specialize the general next-relation N to a next-function resulting in a simpler version of Def. 5.3:

$$\forall s \in \Sigma_2 \cdot P(s) \Rightarrow N_1(f(s)) = f(N_2(s)))$$

Note that this property directly corresponds to the commuting diagram in Figure 1. Hence, when testing if a method N_2^m refines an abstract method N_1^m, we simply initialize both objects to a state such that the refinement mapping f and the invariants hold, then execute both mappings and check if the results are equal under the mapping f.

[7] https://fscheck.github.io/FsCheck/

[8] https://github.com/HypothesisWorks/hypothesis

[9] https://scalacheck.org

```
 1  property("Class C2 refines C1 for method m()") =
 2    forAll(validStates) { object2: Class2 =>
 3      val object1 = refinement_mapping(object2)
 4      if (object1.guards && object2.guards) {
 5        object1.m()
 6        object2.m()
 7        object1 == refinement_mapping(object2)
 8      }
 9      else true
10    }
```

Fig. 2. Scetch of a ScalaCheck property for testing that a method of Class1 correctly refines its abstract version in Class0.

Figure 2 shows how such a refinement property can be realized in ScalaCheck. The property checks for all states of the concrete class `Class2` that a method `m` respects refinement. A custom generator `validStates` randomly generates objects of `Class2` that also respect its invariant (Line 2). Hence, the invariant check is part of this custom generator. Then, the refinement mapping is used to construct or initialize the abstract version `Class1` (Line 3). If the guards of the methods are satisfied, both methods are executed and the equivalence of the resulting object states is checked (lines 4–7).

Note that we may introduce new methods under refinement where no abstract counterpart method exists. In such cases, we refine an abstract skip-method, i.e. we simply need to test that the new method N_2^m (`object2.m`) does not change the abstract state. This latter case is also known as stuttering. In addition, we need test-properties for checking that all methods preserve their (class) invariants.

The bold idea, we are following here, is that we will prompt ChatGPT in order to generate the test properties as well. One may argue that this generation of test-properties introduces another verification problem. Therefore, we need to review the test-properties. Our hypothesis is that these declarative test-properties are easier to review than the generated class implementations (code review vs. specification review).

4 The Bridge Controller Example

In this section we outline the requirements for the system to be developed, a bridge controller originally described in [2]. The objective is to develop a software program, the *controller*, which controls the movement of cars on a bridge connecting a mainland and an island, the *environment*, shown in Figure 3.

Cars can move from the mainland onto the bridge, and from there to the island, and similarly in the other direction. The following requirements are defined in [2], divided into requirements concerned with the functionality of the controller, labeled $\textsc{Fun}_i$, and requirements concerned with the environment, labeled $\textsc{Env}_i$.

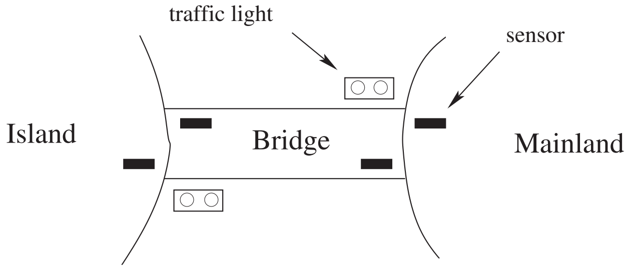

Fig. 3. The bridge environment from [2].

FUN_1**:** *The system is controlling cars on a bridge connecting the mainland to an island.*
FUN_2**:** *The number of cars on bridge and island is limited.*
FUN_3**:** *The bridge is one-way or the other, not both at the same time.*
ENV_1**:** *The system is equipped with two traffic lights with two colors: green and red.*
ENV_2**:** *The traffic lights control the entrance to the bridge at both ends of it.*
ENV_3**:** *Cars are not supposed to pass on a red traffic light, only on a green one.*
ENV_4**:** *The system is equipped with four sensors with two states: on or off.*
ENV_5**:** *The sensors are used to detect the presence of a car entering or leaving the bridge: "on" means that a car is willing to enter the bridge or to leave it.*

Note that FUN_3 describes what is commonly referred to as a *one-lane* bridge. In the following text it will be referred to as a one-way bridge, like in [2]. In [2] the bridge controller is developed in four steps (three refinements), first a mainland and an island (without a bridge), then the bridge is introduced, then the traffic lights, and finally the sensors. In this paper we shall just develop the two first of these, including their refinement relation.

5 Guarded-Commands Development

In this first experiment we aim for developing a Scala program that is as close as possible to Abrial's original Event-B model of the bridge controller [2]. Event-B follows a guarded-command style, i.e. a set of variables is iteratively updated by guarded commands. A guarded command is a conditional assignment that is enabled when the guard-condition is satisfied in a given state. In Event-B these guarded commands are called events. If more than one event is enabled, one is chosen non-deterministically.

In Scala, we can mimic this style with a class of methods with preconditions. Each method resembles an event. Semantically, we assume that these methods are non-deterministically called from the environment. If a precondition is not satisfied, the state does not change and another method is tried. In this way, we can write Event-B-like models in the Scala programming language. The main difference is that we require executable events. In the following, we report about our experience of letting ChatGPT write, refine and test the model.

5.1 Initial Model

Abrial starts with a very abstract model, consisting of two events that increase and decrease the number of cars on the island. In a first prompt, we tell ChatGPT what we want using the variable and method names of Abrial's model.

> **Prompt:** Give me a Scala class Bridge0 that abstractly models an island where at most d cars are allowed. A method ml_in shall model a car going from the island to the mainland. A method ml_out shall model the entering of a car from the mainland. At this level, the model state shall only be a variable n, the number of cars on the island. The following class invariant shall be respected: n shall be positive and not exceeding d and d shall be greater than 0.

The initial result was correct, but the style was not what we aimed for: ChatGPT produced a version with if-conditions, but we want preconditions. Furthermore the counter variable n was initialized to zero, but we prefer a more general parameterized initialisation. This is what we tell ChatGPT.

> **Prompt:** The code compiles. Conditions are used, it would be better to use preconditions. Furthermore, it would be better to make n a variable parameter such that it can be set when the class is constructed.

This led to the following solution satisfying our requirements, which we name GC.V1 (Guarded Commands Version 1).

```scala
class Bridge0(d: Int, private var n: Int = 0) {
  require(d > 0, "d must be greater than 0")

  // Ensure n starts off with a valid value
  require(n >= 0 && n <= d, "Initial number of cars on the island is invalid"
      )

  // method to model the entering of a car from the mainland to the island
  def ml_out(): Unit = {
    require(n < d, "Number of cars on the island exceeds limit!")
    n += 1
  }

  // method to model a car going from the island to the mainland
  def ml_in(): Unit = {
    require(n > 0, "No cars on the island to remove!")
    n -= 1
  }

  def carsOnIsland: Int = n
}
```

GC.V1.bridge

The above code is a parameterized Scala class with the maximum number of cars d and the state variable n being parameters. Note that d is constant while n can be changed after construction. The two methods resemble the two abstract events with guards ensuring that the invariant holds. The require-expressions have an error message to be included in the exception when the condition is violated. Here, also the invariant is encoded via two require-clauses. These clauses are checked upon construction of the object. Note that ChatGPT did a good job in commenting the program.

Next, we want to generate test-cases with ScalaCheck. Therefore, we let ChatGPT generate ScalaCheck properties that can be tested automatically. Obvious properties to be tested are the invariants.

> **Prompt:** Given the class Bridge0, give me a ScalaCheck property that checks if the constructor and methods respect the following invariants: (1) n shall be positive, (2) n shall not exceed d, (3) d shall be greater than 0.

Two Iterations were needed, because ChatGPT generated a property that had several flaws. The code did not compile. Furthermore, the tests failed when a precondition was violated, but such tests should be ignored. In addition, one property did not catch exceptions and another property was useless, in fact, testing nothing. Hence, in the following prompt we communicated these observations in order to correct the flawed property.

> **Prompt:** The last property can be skipped. It also does not compile. There seems to be a misunderstanding: we wanted a property that checks that the maximum allowed number of cars on the island is greater than zero. Furthermore, I think that the property "constructor" shall also test if an exception in case of invalid arguments is raised. Similar to the other two properties.

The result was the following ScalaCheck property.

```
1  object Bridge0Properties2b extends Properties("Bridge0") {
2    // Generator for valid d values (greater than 0)
3    val genD: Gen[Int] = Gen.posNum[Int]
4  ...
5   property("ml_out respects invariants") = forAll(genD) { d: Int =>
6      val bridge = new Bridge0(d, d) // initialize with maximum cars
7      val result = try {
8        bridge.ml_out()
9        bridge.carsOnIsland <= d
10     } catch {
11       case e: IllegalArgumentException => bridge.carsOnIsland == d
12     }
13     result :| s"ml_out with d = $d and n = ${bridge.carsOnIsland}"
14   }
15 ...                                          GC.V1.property-basted-test
```

The above listing shows the correct ScalaCheck property for method `ml_out()`. The code for `ml_in` is similar. First in Line 3, a generator for natural numbers is defined. It is used in the following property (Line 5) to generate values of the maximum number of cars d, i.e. the property should hold for all natural numbers d. Next, in Line 6 a bridge with the maximum number of allowed cars is created. Then, the method is called at Line 8. This call shall raise an exception leaving the number of cars on the island unchanged. Note that this property constitutes a boundary test at the upper limit, initializing the initial number of cars n to the maximum d. When running ScalaCheck it produces 100 random tests and all pass. The same holds for the property of method `ml_in`.

5.2 Refinement

In the following, we will refine the initial Scala model by introducing the one-way bridge going to the island.

Prompt:

– Given the class Bridge0, give me a refined class Bridge1 with three variables:
  - a: the number of cars on the bridge to the island,
  - b: the number of cars on the island,
  - c: the number of cars on the bridge from the island to the mainland.
  - The refinement relation is n = a + b + c.

– We need now two additional methods: il_in for cars going from the bridge to the island and il_out for cars leaving the island and going on the bridge.

– There is a new invariant: the bridge shall be a one-way bridge.

Again, the first attempt was not successful. The preconditions of `ml_out` and `ml_in` violated the one-way rule of the bridge, because ChatGPT mixed up variable names. In addition, the first version did not allow the free initialization of a, b, and c. Formulating these observations in a new prompt created a refined version of the bridge controller.

```
class Bridge1(d: Int, initialA: Int = 0, initialB: Int = 0, initialC: Int =
    0) {
  require(d > 0, "d must be greater than 0")
  require(initialA + initialB + initialC <= d, "Initial number of cars
      exceeds
                                              limit")

  private var a: Int = initialA  // cars on the bridge to the island
  private var b: Int = initialB  // cars on the island
  private var c: Int = initialC  // cars on the bridge from the island to the
                                 // mainland

  // One-way bridge invariants
  private def oneWayInvariant(): Boolean = {
    !(a > 0 && c > 0)
  }
  ...
  // method to model the entering of a car from the mainland to the bridge
  // towards the island
  def ml_out(): Unit = {
    require(c == 0, "Bridge is already in use by cars from the island to mainland")
    require(b + c + a < d, "Number of cars exceeds limit!")
    a += 1
  }
  ...
  require(oneWayInvariant())
}
```

GC.V2.bridge

The above code snippet shows the part of the refined model with the new variables, its invariants and the refined version of `ml_out()`. This class can initialize a, b, c individually, it correctly checks that their sum does not exceed the maximum d, and a new one-way invariant has been introduced and is checked. The preconditions are correct including nicely generated error messages.

The above code seems to be correct. In the following, we attempted to generate a property to test that refinement holds. The following prompt guided ChatGPT.

> **Prompt:** Given the two classes Bridge0 and Bridge1. Generate a ScalaCheck property that checks if ml_out of Bridge1 is a refinement of the same method in Bridge0 using the following refinement relation: n = a + b + c. The generated classes shall respect their invariants, hence all invalid classes shall be filtered out.

In addition, we included the code of Bridge0 and Bridge1 into the above prompt. This time, the results of ChatGPT were less convincing as the generated code included many logical flaws. In total, we needed ten iterations: four iterations in order to get a correct refinement property for method `ml_out` and six iterations for the two new methods for entering and leaving the island from the bridge. The refined version of method `ml_in` having the simplest precondition was immediately correct. The two main issues were (1) sophisticated but slightly wrong class generators that raised exceptions due to invariant violations and (2) precondition violations of methods were wrongly classified as test failures. The latter is an error that also appeared in the first test property for testing the initial model. Finally, after ten attempts, ChatGPT produced a test property for testing that the refinement property holds.

```
...
def genValidBridgeStates(d: Int): Gen[(Bridge0, Bridge1)] = {
  for {
    a <- Gen.choose(0, d)
    b <- Gen.choose(0, Math.max(0, d - a))
    c <- Gen.choose(0, Math.max(0, d - a - b))
    if (a > 0 && c == 0) || (c > 0 && a == 0) || (a == 0 && c == 0)
    bridge0 = new Bridge0(d, a + b + c)
    bridge1 = new Bridge1(d, a, b, c)
  } yield (bridge0, bridge1)
}
...
property("Bridge refinement for ml_out, ml_in, il_in, and il_out") =
forAll(genD) { d: Int =>
  val validBridgeStates = genValidBridgeStates(d)
  Prop.forAllNoShrink(validBridgeStates) { case (bridge0, bridge1) =>
```

```
17
18  // Properties for ml_out
19  val mlOutResult = if (bridge0.carsOnIsland < d && bridge1.carsFromIsland ==
        0) {
20    bridge0.ml_out()
21    bridge1.ml_out()
22    bridge0.carsOnIsland == bridge1.carsToIsland + bridge1.carsOnIsland +
23                            bridge1.carsFromIsland
24  } else true
25  ...
26   mlOutResult && mlInResult && ilInResult && ilOutResult
27                                                        GC.refinement.property
```

The generated property tests that `ml_out` and `ml_in` are correctly refined and that the new methods `il_in`, `il_out` do not change the abstract state, i.e. they refine *skip* (stuttering). Interestingly, ChatGPT came up with a property that combines the test results for all four methods (Line 26). The property initializes the classes such that $a + b + c$ stay below maximum d. It also filters states that violate the one-way invariant. Note that both initialized bridge objects respect the refinement mapping $n = a + b + c$.

6 Object-Oriented Development

In this approach we gave ChatGPT more freedom by not asking for a solution based on guarded commands. Perhaps not surprising, it pursued an object-oriented solution, which we shall describe in this section. There were in fact two attempts, the first of which was abandoned after several prompts. We illustrate both attempts below, the first one only briefly and the second one in more detail. First, we provided ChatGPT with the following prompt.

> **Prompt:** Write a Scala program, which models cars moving between two zones: a mainland and an island connected by a bridge. There are some requirements defined. These are as follows: [FUN1 ... FUN3, ENV1 ... ENV5 provided]. However, we want to develop this program using stepwise refinement where we start with an abstract program that ignores certain details, and then we want to refine the program in steps, adding more and more details. In the first step we ignore the bridge and just consider there to be a mainland and an island. There can only be maximally d cars on the island at any point in time. Initially there are 0 cars on the island. We need an operation for moving a car from the mainland to the island and for moving a car from the island back to the main land. Can you write the Scala 3 program for the first step?

6.1 A First Attempt

A first generated solution was cleverly suggesting a class representing the notion of a *place*, which could then either be the island, the mainland, or the bridge:

```
class Place(var name: String, var carLimit: Int, var currentCars: Int) {
  def addCar(): Boolean = {...}
  def removeCar(): Boolean = {...}
}
```

Although the idea was appealing, the solutions were flawed, needing numerous prompts. Moving a car from a place to another place was placed in the main program and not as part of the `Place` class. If the source place had cars but the target place had no more space, a car will be moved from the source anyway, but not transferred to the target. It generally had problems with the bridge, how many cars to allow and in what directions they could drive, going from allowing only one car on the bridge to allowing cars in both directions, each of which is wrong. Also it moved the wrong number of cars. There were also compiler errors. It was decided to abandon this thread and start over. The following sections cover the second attempt.

6.2 Initial Model

We started over again with the exact same prompt. This led interestingly to a different solution introducing an `Island` class and a `Mainland` class, instead of the generic `Place` class. It, however, took two additional prompts to solve some issues, including some style issues concerning the design of classes (there was initially only an Island class and no Mainland class), and our wish that moving a car should just be one operation, not require two method calls. An error was also detected, that a car is moved to the island even when there are no cars on the mainland. This led to the following self explanatory solution, which we name OO.V1 (Object-Oriented Version 1).

```
class Island(maxCars: Int) {
  private var currentCars: Int = 0

  def canAcceptCar: Boolean = currentCars < maxCars

  def addCarFromMainland(): Boolean = {
    if (canAcceptCar) {
      currentCars += 1
      true
    } else {
      false
    }
  }

  def moveToMainland(): Boolean = {
    if (currentCars > 0) {
      currentCars -= 1
      true
    } else {
      false
    }
  }

  def carsOnIsland: Int = currentCars
}
```

OO.V1.island

```scala
class Mainland {
  private var cars: Int = 0

  def addCar(): Unit = {
    cars += 1
  }

  def moveToIsland(island: Island): Boolean = {
    if (cars > 0 && island.canAcceptCar) {
      cars -= 1
      island.addCarFromMainland()
      true
    } else {
      false
    }
  }

  def carsOnMainland: Int = cars
}
```

OO.V1.mainland

We also asked ChatGPT to rename `Island.addCarFromMainland()` as `Island.addCar()` since it really just adds a car to the island. We then proceeded to ask ChatGPT to generate a property-based test with the following prompt.

> **Prompt:** Can you write tests in ScalaCheck, using property-based testing, which test these classes?

This first resulted in a solution containing the following code.

```scala
  property("moveToMainland transfers car from island to mainland") =
  forAll { (initialMainlandCars: Int, initialIslandCars: Int) =>
    ...
    if (beforeIsland > 0) {
      (afterMainland == beforeMainland) && (afterIsland == beforeIsland - 1)
    } else {
      ...
    }
  }
```

As can be seen, this is not quite correct in that it only checks that a car is removed from the island, and not that it is added to the mainland. In fact, this corresponds to ChatGPT's generated explanation of the solution.

> **Response:** ... If you move a car from the mainland to the island, the number of cars on the mainland should decrease by one, and the number on the island should increase by one, unless the island is full or the mainland has no cars. **If you move a car from the island to the mainland, the number of cars on the island should decrease by one, unless the island has no cars**.

The generated English language requirement is wrong with respect to moving a car from the island to the mainland by not requiring the car to be added to the mainland, and the code generation "correctly" follows that. A second issue was that it generated negative numbers and very large positive numbers as test inputs, which caused the test to run for a long time. Finally for the amusement, we prompted it with *“Is the ’foreach’ construct the preferred Scala3 way to write a for loop?”* (finding it slightly unappealing), to which it responded: *“foreach is a method on collections and is still idiomatic for iterating over collections and performing side-effects”*, sticking to its guns. The final result after four prompts was as follows, including here only the test of moving a car from the island to the mainland.

```
1  object CarSystemSpec extends Properties("CarSystem") {
2    val maxCarsOnIsland = 5
3    val limitedGen = choose(0, 10)
4    ...
5    property("moveToMainland transfers car from island to mainland") =
6    forAll(limitedGen, limitedGen) {
7    (initialMainlandCars: Int, initialIslandCars: Int) =>
8      val mainland = new Mainland()
9      val island = new Island(maxCarsOnIsland)
10     (0 until initialMainlandCars).foreach(_ => mainland.addCar())
11     (0 until initialIslandCars).foreach(_ => island.addCarFromMainland())
12
13     val beforeMainland = mainland.carsOnMainland
14     val beforeIsland = island.carsOnIsland
15     island.moveToMainland()
16     val afterMainland = mainland.carsOnMainland
17     val afterIsland = island.carsOnIsland
18
19     if (beforeIsland > 0) {
20       (afterMainland == beforeMainland + 1) && (afterIsland == beforeIsland -
            1)
21     } else {
22       (afterMainland == beforeMainland) && (afterIsland == beforeIsland)
23     }
24   }
25 }
```

OO.V1.property-basted-test

As can be seen it has switched to use a “user defined” generator (Line 3) as a response to the prompt on negative and too large positive numbers. It first initializes the island and the mainland (lines 8-11). It then stores the pre-values of cars on the mainland and island (lines 13-14). Then it moves a car (Line 15). Then it obtains the post-values of cars on the mainland and island (lines 16-17). The condition itself checks that a car is moved in case there were cars on the island, or else nothing is changed (lines 19-23).

When running the test, however, it failed with a message from ScalaCheck, which we directly fed back to ChatGPT as a prompt without any further explanations.

Prompt:

```
! CarSystem.moveToMainland transfers car from island to mainland:
  Falsified after 0 passed tests.
> ARG_0: 0
> ARG_1: 1
> ARG_0_ORIGINAL: 10
> ARG_1_ORIGINAL: 8
Found 1 failing properties.
```

The error is caused by the fact that `island.moveToMainland()` does not actually add a car to the mainland. What is interesting is that ChatGPT itself corrects the error based on ScalaCheck's error message as a prompt, suggesting that one can imagine an automated feedback loop between a testing tool and an LLM. The corrected part of the code is shown below where a reference to the mainland has been added as parameter to the `moveToMainland` method (Line 3), and a `mainland.addCar()` statement has been added (Line 6).

```
1  class Island(maxCars: Int) {
2    ...
3    def moveToMainland(mainland: Mainland): Boolean = {
4      if (currentCars > 0) {
5        currentCars -= 1
6        mainland.addCar()
7        true
8      } else {
9        false
10     }
11   }
12   ...
13 }
```

OO.V1.island-corrected

6.3 Refinement

At this point we move forward and ask ChatGPT to perform a refinement with the following prompt.

> **Prompt:** Now let's do the first refinement. We introduce a bridge between the mainland and the island. The bridge is a one way bridge in the sense that cars can only go in one direction at a time. We now need four operations: (1) for moving a car from the mainland to the bridge, (2) for moving a car from the bridge to the island, (3) for moving a car from the island to the bridge, and (4) for moving a car from the bridge back to the mainland. We do not yet introduce traffic lights.

There were several obstacles on the way to a working solution. There were some style issues, including again two method calls to make a move, inconsistent ways of checking whether a move is possible (in a condition: asking for forgiveness after a failed attempt versus asking for permission), naming of methods, and code that could be simplified. There was a syntax error in a private name not being accessible. There were also semantic errors, including that a car did not get added to mainland. There was wrongly also no limit to how many cars can be on the bridge. The Fun$_2$ requirement stated: *The number of cars on bridge and island is limited*. This requirement was admittedly not quite precise. To make it precise, we required that in order to move a car from the mainland to the island, the sum of cars on the island and on the bridge going in the direction of the island should be less than the maximal allowed number of cars on the island. Further errors included that cars could go in both directions at the same time, or that it only allowed one car on the bridge at a time, and that the maximal number of cars on bridge was defined as constant rather than a function of cars on island.

Below is the final result after seven prompts, showing the implementation of the bridge, the island, and the mainland. First the `Bridge` class together with an enumerated type of directions.

```
 1  enum Direction:
 2    case ToIsland, ToMainland
 3
 4  class Bridge(island: Island, mainland: Mainland) {
 5    private var _carsOnBridge: Int = 0
 6    private var direction: Direction = Direction.ToIsland
 7
 8    def canAcceptCar(d: Direction): Boolean = {
 9      if (d == Direction.ToIsland) {
10        return _carsOnBridge + island.carsOnIsland < island.maxCars &&
11          (_carsOnBridge == 0 || direction == d)
12      }
13      _carsOnBridge == 0 || direction == d
14    }
15
16    def moveCarOntoBridge(d: Direction): Unit = {
17      _carsOnBridge += 1; direction = d
18    }
19
20    def moveToIsland(dest: Island): Boolean = {
21      if (_carsOnBridge > 0 && direction == Direction.ToIsland && dest.
            canAcceptCar)
22      {
23        _carsOnBridge -= 1; dest.acceptCar(); true
24      } else { false }
25    }
26
27    def moveToMainland(): Boolean = {...}
28  }
```

OO.V2.bridge

The `Bridge` class above is relatively straightforward, except perhaps for the condition determining whether a car can be moved onto the bridge in some direction d, defined by the method `canAcceptCar` (lines 8-14). If the direction is towards the island, the number of cars on the island plus the current number of cars on the bridge should be less than the allowed maximum number of cars on the island. In any case, either there should be no cars on the bridge or the direction d must match the current direction of the other cars. The classes `Island` and `Mainland` below are relatively straightforward.

```
class Island(val maxCars: Int) {
  private var currentCars: Int = 0

  def canAcceptCar: Boolean = currentCars < maxCars

  def carsOnIsland: Int = currentCars

  def moveToBridge(bridge: Bridge): Boolean = {
    if (currentCars > 0 && bridge.canAcceptCar(Direction.ToMainland)) {
      bridge.moveCarOntoBridge(Direction.ToMainland)
      currentCars -= 1
      true
    } else {
      false
    }
  }

  def acceptCar(): Unit = {
    currentCars += 1
  }
}
```

OO.V2.island

```
class Mainland {
  private var cars: Int = 0

  def addCar(): Unit = cars += 1

  def carsOnMainland: Int = cars

  def moveToBridge(bridge: Bridge): Boolean = {
    if (cars > 0 && bridge.canAcceptCar(Direction.ToIsland)) {
      bridge.moveCarOntoBridge(Direction.ToIsland)
      cars -= 1
      true
    } else {
      false
    }
  }
}
```

OO.V2.mainland

It was interesting to observe that the object-oriented design distributes the logic in a manner that makes it more difficult to ensure oneself that the code is correct compared to the guarded command approach. E.g. to understand how a car is moved from the island to the bridge one has to study several methods distributed over the two classes. In the guarded command solution the entire action is defined in one place.

We have now developed two versions of our bridge control system, an abstract initial version OO.V1 and a refinement OO.V2. We first performed a property-based test of OO.V2, which we will not report upon in detail here due to lack of space. It did not reveal any errors in V2. However, it is worth mentioning that it took 10 prompts to get the property-based test itself right. The main issue was that it repeatedly did not initialize all of mainland, bridge, and island correctly with an initial random (in ScalaCheck style) number of cars, it only initialized the place moved from. What is more interesting is that we fed the failed test outputs as prompts (as shown earlier) repeatedly without additional explanations. However, this process did not seem to converge, and ChatGPT needed an additional small hint at the end before success. After this property-based testing exercise we then proceeded to the actual refinement test with the following prompt.

Prompt: The task is now to show that the second version V2 indeed is a refinement of the first version V1. This is done by first defining a mapping M from the state of V2 to the state of V1, and then show that given a reachable state S2 of V2, if we apply an action A2 of V2 and reach a state S2', then if we apply the corresponding action A1 in V1 to M(S2) we get a state S1' such that S1' = M(S2'). Can you write a test using ScalaCheck that performs this test? I will give give you the two versions V1 and V2 of the code [... the code of V1 and V2 ...].

Generating a refinement test was non-trivial and required 56 prompts, involving the following issues, amongst others. The initial test moved several cars from mainland to island over the bridge before testing the intermediate states. It should check one move at a time. It also initially tested only one direction. Also, at some point the condition verified was wrong, verifying that all cars on the bridge got moved to the island in one move. In another attempt, the test for moving a car from the bridge in V2 to the island compared this to moving a car from the island in V1 to the mainland (wrong direction). It also made wrong subtractions and additions. Some iterations were spent ensuring that the concrete version only made moves from the bridge to the mainland or island if the direction was correct. At some point it wanted to compare island (and mainland) objects with equality (`==`). This was an interesting suggestion, but it had not defined equality (the `equals` method), and this would also cause a refactoring of the code, which we at this point tried to avoid.

At some point it produced the following code for moving a car from the mainland to the bridge in the concrete version V2.

```
1 val movedToBridge = mainlandV2.moveToBridge(bridge)
2
3 val (mainlandV1, islandV1) = mapStateV2toV1(mainlandV2, islandV2, bridge)
4
5 movedToBridge == mainlandV1.moveToIsland(islandV1)
```

First, a move is performed at the concrete V2 level (Line 1), storing the Boolean result (success or failure) in the variable `movedToBridge`. The *resulting* concrete V2 state is then mapped to the corresponding abstract state (Line 3). Finally, a move is performed at the abstract V1 level, starting from that abstracted state, and the result is asserted equal to the result of the concrete move, stored in the variable `movedToBridge` (Line 5).

There are a few errors here. First of all, it maps the post-state of the V2 move to V1, whereas it should have been the pre-state such that V1 can make a move from that. Second, it compares the move from mainland to the bridge in V2 to a move from the mainland to the island in V1, and these yield different results. Note that in V2, a car on the bridge going in the direction of the island is considered as still belonging to the mainland in V1, so in V1 no car should be moved to the island. In general, it had a hard time learning that cars on the bridge moving to the island in V2 belong to the mainland in V1, and similarly the other way, cars on the bridge moving to the mainland in V2 belong to the island in V1.

This is related to the problem of *stuttering* in V1. This is the situation where an action in the concrete version V2 does not correspond to an action in V1. This is normally in refinement proofs handled by allowing the abstract version, here V1, to stutter, to make a move that does not change its state. Stuttering was initially not modeled in the generated test code, which, however, interestingly was caught by the refinement test. Specifically, in the property above a move from mainland to the bridge in V2 is attempted compared to a move between mainland and island in V1, whereas it should have been a stuttering step in V1. We informed ChatGPT about stuttering with the following prompt.

Prompt: The [problem] is “stuttering”: stuttering is the situation where an action in the concrete version (V2 here) does NOT have a corresponding action in the abstract version. In this case, the starting state and the result state of the concrete action map to the same abstract state. We say that the action in the concrete version corresponds to a stuttering action in the abstract version (that does not change the state). So in the first property, mainlandV2.moveToBridge causes a car to be moved from the mainland to the bridge but since cars on the bridge going to the island are counted as belonging to the mainland in the abstract version, this corresponds to no action in the abstract state.

A different problem was related to the generator of V2 states, caught by a failing test. The error was caused by the refinement mapping from V2 to V1, which only adds to the island if there is space. The mapping calls `islandV1.addCar()`, which contains the check: `if (canAcceptCar)...`. This means that not all cars that are on the island and on the bridge in Mainland direction in V2 are necessarily added to the abstract island by the mapping if there are “too many” cars. The concrete state should be initialized such that the sum of cars on the island and on the bridge going towards the mainland is less than or equal the maximum number of cars allowed on the island. In the end we manually had to fix the generator of V2 states.

At several stages it wanted to change the V2 implementation to have the direction be part of the implementation. However, considering how fragile ChatGPT is it was decided not to follow that advice, trying to avoid redoing too much work. It was also necessary to remind it of what versions V1 and V2 were. It also changed the properties to test even when they were ok, and it had to be asked to go back to previous definitions, reminding it what they were. These are issues that likely would be avoided with a system such as Copilot.

The following refinement test using ScalaCheck was finally generated. First it generated the following generator of inputs.

```
1  object RefinementProperties extends Properties("Refinement") {
2    val genCars = for {
3      m <- Gen.choose(0, 10)
4      maxIslandCars <- Gen.choose(0, 10)
5      i <- Gen.choose(0, maxIslandCars)
6      d <- Gen.oneOf(Direction.ToMainland, Direction.ToIsland)
7      b <- Gen.choose(0, maxIslandCars - i)
8    } yield (m, b, i, maxIslandCars, d)
9    ...
10 }
```

OO.refinement.generator

What is of interest here is the number of cars generated for the bridge (Line 7), taking into account that the number of cars on the island and bridge should reflect (not surpass) the maximum number of cars allowed on the island. This required numerous prompts.

Next, the refinement mapping is defined by the `mapStateV2toV1` method shown below.

```
1   def mapStateV2toV1(mainlandV2: MainlandV2, islandV2: IslandV2,
2                      bridge: Bridge): (MainlandV1, IslandV1) =
3   {
4     val mainlandV1 = new MainlandV1()
5     val islandV1 = new IslandV1(islandV2.maxCars)
6
7     (1 to mainlandV2.carsOnMainland +
8     (if (bridge.getDirection == Direction.ToIsland) bridge.carsOnBridge else
          0))
9       .foreach(_ => mainlandV1.addCar())
10
11    (1 to islandV2.carsOnIsland +
12    (if (bridge.getDirection == Direction.ToMainland) bridge.carsOnBridge
          else 0))
13      .foreach(_ => islandV1.addCar())
14
15    (mainlandV1, islandV1)
16  }
```

OO.refinement.mapping

It takes as argument a mainland, an island, and a bridge of version OO.V2 and returns a tuple consisting of the corresponding mainland and island of version OO.V1. It first creates new instances of OO.V1 mainland and island (lines 4-5). It then adds cars to the mainland (lines 7-9). That is, for each car in the OO.V2 mainland plus the cars on the bridge going in direction of the island, it adds a car to the OO.V1 mainland. Similarly, it adds cars to the OO.V1 island that are on the bridge going in the direction of the mainland (lines 11-13), in addition to the cars on the OO.V2 island. At the end, it returns the updated OO.V1 mainland and island (Line 15).

Finally, the refinement property, corresponding to Definition 5.3, for moving a car from the bridge to the island is shown below.

```
1   property("Bridge to Island move is consistent") = forAll(genCars) {
2     case (m, b, i, maxIslandCars, d) =>
3       val mainlandV2 = new MainlandV2()
4       val islandV2 = new IslandV2(maxIslandCars)
5       val bridge = new Bridge(islandV2, mainlandV2)
6       (1 to m).foreach(_ => mainlandV2.addCar())
7       (1 to b).foreach(_ => bridge.moveCarOntoBridge(d))
8       (1 to i).foreach(_ => islandV2.acceptCar())
9
10      val (mainlandV1, islandV1) =
11        mapStateV2toV1(mainlandV2, islandV2, bridge)
12
13      val movedToIslandV2 = bridge.moveToIsland(islandV2)
14      val movedToIslandV1 =
15        if (movedToIslandV2) mainlandV1.moveToIsland(islandV1) else false
16
17      val (mainlandV1Post, islandV1Post) =
18        mapStateV2toV1(mainlandV2, islandV2, bridge)
19
20      (movedToIslandV2 == movedToIslandV1) &&
21        mainlandV1Post.carsOnMainland == mainlandV1.carsOnMainland &&
22        islandV1Post.carsOnIsland == islandV1.carsOnIsland
23  }
```

OO.refinement.property

It first initializes a OO.V2 state consisting of a mainland, an island, and a bridge (lines 3-8). It then maps this concrete OO.V2 state to an OO.V1 state using the refinment mapping (Line 10). It then performs the moves, a concrete on OO.V2 and an abstract on OO.V1 if there was a concrete move (lines 13-15). Then it maps the resulting OO.V2 state to a OO.V1 state using the refinement mapping

(Line 17). At the end, it checks the commuting diagram condition, that the OO.V1 state obtained by the abstract move is the same as the state obtained by mapping the resulting OO.V2 state to a OO.V1 state using the refinement mapping (lines 20-22).

7 Conclusion

The developments using an LLM like ChatGPT turned out not to be effortless. This is likely due to the fact that the problem being solved is not a common pattern occurring in open source software. We let ChatGPT generate all the code (except for a couple of exceptions), including test properties, since we tried to see how far this approach could be taken. In some cases ChatGPT went off in a wrong direction and had to be steered back on the right track. The experience was different than the *"how do you do X in programming language Y"* prompts that ChatGPT is so good at.

It turned out that the object-oriented approach required almost 5 times as many prompts (80) than the guarded command approach (17). There can be several reasons for that but we did observe that it was easier to convince oneself that a guarded command program was correct than an object-oriented program. We attribute this to the fact that in an object-oriented solution, a move (e.g. from bridge to mainland) was distributed over several methods in different classes. This aspect could potentially impact the performance of ChatGPT as well.

One learns to write better requirements through an effort like this, what is also referred to as *prompt engineering*. One can imagine requirements being translated to a formalism, as in our case, which can then be checked for soundness and completeness. Once the generated artifact is "good enough", the requirements are possibly also "good enough".

However, what required the most effort was to read and comprehend the generated code, since many prompts lead to many results that have to be investigated, which was quite tiresome. ChatGPT does not point out (in the code) what has been changed, so we had to read all the generated code after each prompt. A system integrated in an IDE, such as Copilot, should do a better job here. As it turned out, it was easier to read and understand the generated tests, and in particular the property-based tests. This suggests that tests, and in particular succinct and readable specifications, become important in evaluating generated code.

Our process consisted, for each result, of writing down observations, and then re-formulating them as a prompt. In case of failed test cases we first tried to feed them back directly as prompts without modification. As we showed, this worked in some cases in the sense that ChatGPT "understood" what the problem was and fixed it. In other cases this approach did not converge. Since the process involved prompt engineering in natural language (English), the results shall be interpreted with a grain of salt. Prompt engineering is an informal process. Nonetheless, we believe that we can conclude that testing and specification are important concepts in this context.

Wrt. future work, a challenge for research in this area is the unknown nature of future improvement of LLMs' capabilities. If these models become significantly better, research done now may quickly become obsolete (depending on the kind of research of course). It will be interesting to carry out the same experiment with Version 5 of ChatGPT, which is expected to be launched within months of the time of writing, as well as with other LLM models. It would also be interesting to carry out the same experiment using an LLM IDE such as Copilot. One feature that we were missing during the work was access to a tree-like structure of the development branches (perhaps supported by a github), where one easily can "walk" up and down the tree of experiments.

References

1. M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82:253–284, 1991.
2. J. Abrial. *Modeling in Event-B - System and Software Engineering.* Cambridge University Press, 2010.
3. W. Ahrendt, D. Gurov, M. Johansson, and P. Rümmer. TriCo – triple co-piloting of implementation, specification and tests. In T. Margaria and B. Steffen, editors, *ISoLA'22: Leveraging Applications of Formal Methods, Verification and Validation. Verification Principles*, volume 13701 of *Lecture Notes in Computer Science*, pages 174–187. Springer, 2022.
4. B. K. Aichernig and R. Schumi. Property-based testing of web services by deriving properties from business-rule models. *Softw. Syst. Model.*, 18(2):889–911, 2019.
5. T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *Software Testing, Verification and Validation Workshops (ICSTW), 2015 IEEE Eighth International Conference on*, pages 1–4, April 2015.
6. R.-J. Back and J. Wright. *Refinement Calculus - A Systematic Introduction.* Texts in Computer Science (TCS), 1998.
7. L. Belzner, T. Gabor, and M. Wirsing. Large language model assisted software engineering: Prospects, challenges, and a case study. In B. Steffen, editor, *AISoLA'23: Bridging the Gap Between AI and Reality*, volume 14380 of *Lecture Notes in Computer Science*, page 355–374. Springer, 2023.
8. S. Bensalem, C.-H. Cheng, W. Huang, X. Huang, C. Wu, and X. Zhao. What, indeed, is an achievable provable guarantee for learning-enabled safety-critical systems. In B. Steffen, editor, *AISoLA'23: Bridging the Gap Between AI and Reality*, volume 14380 of *Lecture Notes in Computer Science*, pages 55–76. Springer, 2023.
9. D. Busch, G. Nolte, A. Bainczyk, and B. Steffen. ChatGPT in the loop: A natural language extension for domain-specific modeling languages. In B. Steffen, editor, *Bridging the Gap Between AI and Reality*, pages 375–390, Cham, 2024. Springer Nature Switzerland.
10. K. Claessen and J. 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, pages 268–279, New York, NY, USA, 2000. ACM.
11. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM*, 18:453 – 457, 1975.

12. K. Havelund and N. Shankar. A refinement proof for a garbage collector. In E. Bartocci, R. Cleaveland, R. Grosu, and O. Sokolsky, editors, *From Reactive Systems to Cyber-Physical Systems, Essays Dedicated to Scott A. Smolka on the Occasion of His 65th Birthday*, volume 11500 of *Lecture Notes in Computer Science*, page 73–103. Springer, 2019.
13. C. B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, Hemel Hempstead, UK, second edition, 1990.
14. L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, may 1994.
15. L. Lampropoulos and K. F. Sagonas. Automatic WSDL-guided test case generation for PropEr testing of web services. In J. Silva and F. Tiezzi, editors, *Proceedings 8th International Workshop on Automated Specification and Verification of Web Systems*, volume 98 of *EPTCS*, pages 3–16, 2012.
16. C. Morgan. *Programming from Specifications*. Prentice Hall, 1990.