Property-Based Testing in the Context of DB Interactions

Lars Jürgensen

Master's Thesis

Programming Languages and Compiler Construction Group
Faculty of Engineering
Christian-Albrecht University of Kiel
Germany
2021

Advised by
Prof. Dr. Michael Hanus
Dr. Nikita Danilenko

Selbstständigkeitserklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel,

Abstract

Thoroughly testing applications with database interactions is challenging, since the tested functionality depends on the input and the current database state. The goal of this thesis is to discuss different techniques and approaches to test database applications using property-based testing. One approach is to use extended generators, which encapsulate both the generation and insertion of the generated values into the database. This approach is highly flexible and composable. Since it often requires much manual effort to write generators for relational database content, we propose a graph-based algorithm, which generates database content in a demand-driven fashion. We restrict the generated content to be acyclic and prove that all connected, acyclic contents can be generated using this algorithm. We implement the algorithm in a project-independent library in Scala. The implementation is well integrated with the property-based testing library ScalaCheck, as it can be arbitrarily combined with other generators and can be used to define properties. An alternative approach is model-based testing, where a random sequence of operations is executed on the system under test and the observed behaviour is compared to a simplified, abstract model of the system. We propose different techniques to use our graph-based algorithm to generate initial database contents for this model-based testing approach.

Acknowledgements

I would like to express my gratitude to Nikita Danilenko, the main supervisor of this thesis, for his patience, for the numerous insightful conversations, and for giving me advice and guidance during this thesis. I wish to thank *Cap3 GmbH* for supporting the thesis and providing a comfortable place to work. Furthermore, I am grateful for the supervision and feedback from Michael Hanus and for giving me the opportunity to write this thesis in his research group. Finally, I would like to acknowledge Lorenz Boguhn and Nowzar Tasslimi for their helpful feedback on drafts of this thesis and John Hughes from the Chalmers University for some initial recommendations and ideas.

Contents

Chapter 1

Introduction

Thoroughly testing applications with database interactions can be challenging. To test the functionality of a method, it is not only necessary to choose a suitable input, but also to bring the database into a specific state beforehand. Additionally, it is not always sufficient, to verify the output of the method. If the method inserts or mutates database content, techniques to verify these changes to the database are also required. Conventional testing approaches are often example-based: The tester chooses a set of interesting input examples and checks if the behaviour of the system under test matches their expectation. A drawback of this strategy is that the test quality is limited by the thoroughness and the creativity of the tester, in the sense that corner cases are only tested if such tests are explicitly specified. The QuickCheck library for Haskell introduced an alternative approach called property-based testing, where abstract properties of the functionality are defined, which shall hold for any given input [Claessen and Hughes, 2000]. Afterwards, the testing framework calls the functions with randomly generated input, to verify whether the specified properties hold. Initially, property-based testing was meant to test pure scenarios, which means that the tested functions are free of side effects and deterministic such that the same input always leads to the same output. However, applications with database interactions are generally not pure: They contain side effects in the form of read and write operations to the database, and the behaviour does not only depend on the input, but also the current database state. This complicates testing database applications using the property-based testing approach. The goal of this thesis is to examine how this testing technique can still be applied to database applications.

We present the first approach, which we call *writer approach*, in Chapter 3. The idea is to create custom extended generators, which do not only encapsulate the generation of values, but also the insertion of the generated values into the database. This approach is very flexible, and the tester has high control over if, how, and when values are inserted into the database. An advantage of this approach is its compositionality, as complex generators can be defined by com-

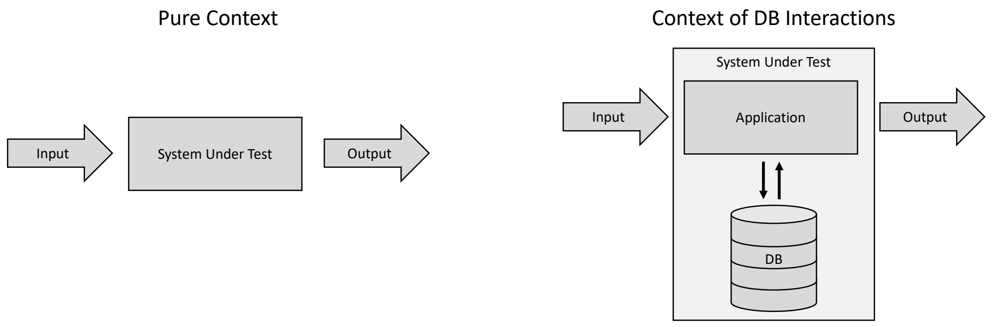

Figure 1.1: A comparison of a pure context and the context of database interactions. In a pure context, the output only depends on the input. In the context of database interactions, there are side effects, and the output depends on the current database state.

bining multiple previously defined generators. However, there are scenarios in which the tester still has to do a lot of manual work to avoid that rare corner cases are missed.

In Chapter 4 we present an approach in which a reference graph is generated. The idea is to randomly generate database content in a demand driven fashion with a graph-based approach. The reference graph generator is implemented using the custom extended generators and, therefore, is a special case of the first approach. We restrict the generated database content to be acyclic. As it is important not to miss corner cases, we prove that all acyclic, connected database contents can be generated by this generator. We implement the proposed algorithm in Scala in a project-independent library. As a proof of concept, we use our implementation to generate data in tests for a real-world application and another small example project.

In Chapter 5 we discuss an already existing approach called *stateful property-based testing*. The idea is not to test the methods (called commands) of the application individually, but to treat the application as a stateful system, and test all commands collectively. For this purpose, the testing framework generates a random sequence of commands and executes these commands on the system under test. Afterwards, the behaviour of the system under test is compared to a simplified abstract model of the application. This approach is suitable for stateful systems in general, and therefore not limited to database applications. We use this approach to test two real-world applications and discuss our experiences and challenges we encountered. Additionally, we propose a hybrid approach, where initial database content is generated using the reference graph generator, and a random sequence of commands is executed afterwards.

In Chapter 2 we introduce the concepts and tools this thesis is founded on. Related work is presented and discussed in Chapter 6. In Chapter 7 we conclude this thesis and present ideas for future work. The index on page 75 shows where specific terms or concepts are introduced.

Chapter 2

Background

In this chapter, we introduce some tools and concepts, which are required to comprehend the topics of this thesis. In Section 2.1 we introduce the programming language Scala. The idea of property-based testing and the ScalaCheck library is presented in Section 2.2. In Section 2.3 we explain the concept of stateful property-based testing, and we list mathematical notations in Section 2.4.

2.1 Scala

All our implementations are written in Scala. Scala is a programming language which combines the object-oriented and functional paradigms [Odersky et al., 2004]. However, the concepts and approaches we propose are mostly independent of the programming language and can be implemented in other languages similarly. Nevertheless, we introduce some basic Scala syntax to make the code examples more understandable. In Scala, methods can be defined using the **`def`** keyword:

```
def add(a: Int, b: Int): Int = a + b
```

The parameters are listed in parentheses and the type of a method, parameter, or variable is defined using colons. Polymorphism can be expressed by adding type parameters in square brackets:

```
def concat[A](a: List[A], b: List[A]): List[A] = a ++ b
```

Scala also allows higher order functions. A function can be passed as a parameter just like other values:

```
def double(value: Int): Int = value * 2
val doubleList = List(1,2,3).map(double)
```

Alternatively, it is also possible to define an anonymous function inside the parameter list:

```
val doubleList = List(1,2,3).map {
    number: Int =>
        number * 2
}
```

2.2 Property-Based Testing

In conventional unit tests, a function is normally called with a fixed input and the output is compared to the expected result:

```
reverse(List(1,2,3)) == List(3,2,1)
```

A disadvantage of this approach is that the functionality is only tested for the explicitly stated inputs. The QuickCheck library introduced an alternative approach, where abstract properties of the tested functionality are defined and afterwards tested by randomly generating test cases [Claessen and Hughes, 2000]. This approach is called property-based testing. We use ScalaCheck, which is a property-based testing library for Scala [Nilsson, 2014]. General properties of the tested function have to be specified, which describe the relationship between the input and output. We use a property of the `reverse` function for lists as an example:

```
val prop = forAll {
    (xs: List[Int], ys: List[Int]) =>
        reverse(xs ++ ys) == reverse(ys) ++ reverse(xs)
}
```

When the property is tested, the lists `xs` and `ys` are randomly generated, and the framework checks whether the property holds for the generated values. By repeatedly testing a property with random inputs, one gains higher confidence in the correctness of the implementation of the function for which the property needs to hold.

```
scala> prop.check
+ OK, passed 100 tests.
```

In the example above, ScalaCheck handles the list generation. However, it is also possible to manually define generators to restrict the generated values, or to generate values of custom types. As an example, we look at how a generator for a user can be implemented.

```
val userGen: Gen[User] = for {
    id <- Gen.uuid
    name <- Gen.alphaStr
    age <- Gen.choose(18, 80)
} yield User(id, name, age)
```

The user generator is implemented by combining three predefined generators from the ScalaCheck library. These generators are combined using a construct called for-comprehension, which is comparable with the **`do`** notation in Haskell.

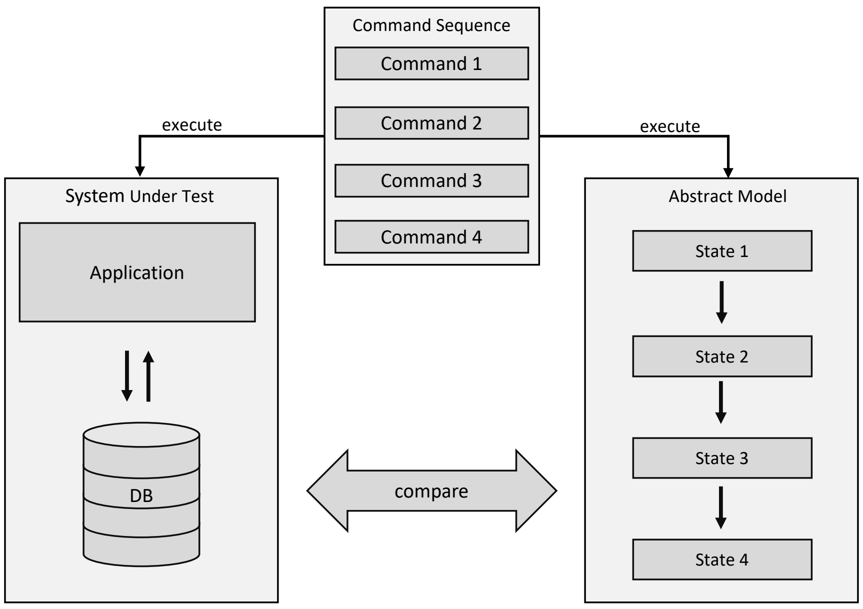

Figure 2.1: Illustration of the stateful property-based testing approach. The test framework generates a random sequence of commands. These commands are executed on the system under test and on a simplified abstract model of the system. Afterwards, the behaviour of the system under test is compared to the abstract model.

2.3 Stateful Property-Based Testing

Property-based testing is especially suitable to test pure functions, which are functions without side effects, where the output solely depends on the input. However, many real-world applications have an internal state, and the behaviour depends on previously called methods. Stateful property-based testing is an approach to use property-based testing to test the behaviour of a collection of stateful methods with interdependencies. These methods are called commands. Instead of only generating the input for a single command, the framework generates a random sequence of commands. As the expected behaviour of each command depends on the previously called commands, an abstract model of the application must be specified by the tester. After a command was executed, the behaviour can be compared to the behaviour of the abstract model of the application. Additionally, one can check if invariants of the application, which should always be satisfied, are still fulfilled.

To demonstrate how a stateful property-based test can be implemented, we introduce an example, where we test an ATM implementation with ScalaCheck. The example only serves the purpose of explaining the concepts, and therefore we omit some required methods. It is based on a more detailed example from the ScalaCheck user guide[1].

```
class ATM {
  def withdraw(amount: Int) = ???

  def deposit(amount: Int) = ???

  def getBalance() = ???
}
```

In this example, the system under test (SUT) is the `ATM` class, and the current state of the abstract model can be implemented with an integer, which represents the current balance:

```
type Sut = ATM
type State = Int
```

Next, we have to define, how an initial state of the abstract model can be generated. In our case, we can always start with a balance of 0:

```
def genInitialState: Gen[State] = Gen.const(0)
```

Each command requires the definition of the following four methods:

- The `run` method defines how the command can be executed on the system under test.
- The `nextState` method specifies how the abstract state changes when the command is executed. If, for example, some money is deposited, the same amount has to be added to the abstract state.
- The `preCondition` defines, under which circumstances a command may be executed. For example, we can specify that money can only be withdrawn, if the current account value is larger than the withdrawn amount.
- The `postCondition` method compares the output of the system under test with the expected behaviour. For example, the output of the `getBalance` method should be equal to the current balance of the abstract model.

We implement the commands `Deposit` and `GetBalance` in the following fashion:

```
case class Deposit(amount: Int) extends UnitCommand {
  def run(atm: Sut): Unit = atm.deposit(amount)

  def nextState(state: State): State = state + amount

  def preCondition(state: State): Boolean = true

  def postCondition(state: State, success: Boolean): Prop = success
}
```

[1]https://github.com/typelevel/scalacheck/blob/main/doc/UserGuide.md

```
case object GetBalance extends Command {
  type Result = Int

  def run(atm: Sut): Result = atm.getBalance

  def nextState(state: State): State = state

  def preCondition(state: State): Boolean = true

  def postCondition(state: State, result: Try[Result]): Prop = {
    result == Success(state)
  }
}
```

The `Deposit` command extends `UnitCommand`, which means that the execution of the `Deposit` command does not return anything. Therefore, the `Result` type is not specified, and the type signature of the `postCondition` method looks slightly different.

When the framework executes the test, it generates a random sequence of commands, so that all preconditions hold. Afterwards, all commands are executed on the system under test. For each command, the output is compared to the expected behaviour using the `postCondition` method. The test is successful if all postconditions are satisfied. If one of the postconditions is not satisfied, the framework tries to minimize the command sequence by iteratively removing commands from the sequence and checking if the smaller test case still leads to failing postconditions. This process is called shrinking and has the goal to find a potentially minimal counterexample.

A risk of stateful property-based testing is that the implementation of the abstract model can be very similar to the actual application and the same bugs are contained in the abstract model. There are several options to avoid this problem:

- A simpler implementation is used, which cannot be used for the actual application due to non-functional requirements. The abstract model can for example use a slower algorithm.
- Only a subset of the functionality is implemented.
- Different technologies are used. For example, when the actual application uses SQL queries, the abstract model can be implemented with in-memory collections of the relevant values.

This testing strategy is particularly useful when what the code should do is simple, but the implementation is complex [Hebert, 2019]. The abstract model can also be implemented in a different programming language than the system under test [Arts et al., 2015].

2.4 Mathematical Notations

In this section, we list mathematical notations used in this thesis.

- The natural numbers $\mathbb{N}$ start with 0. The set of positive integers, which excludes 0, is denoted as $\mathbb{N}_{\geq 1}$.
- Given a set A and a number $n \in \mathbb{N}$, the cartesian power A^n is defined as

$$A^n := \{ (a_1, \dots, a_n) \mid \forall i \in \{ 1, \dots, n \} : a_i \in A \} .$$

- For every number $n \in \mathbb{N}$, the list of pairs

$$((a_1, b_1), (a_2, b_2), \dots, (a_n, b_n))$$

  is written as $((a_i, b_i))_{i=1}^{n}$.
- Let $f \colon A \to B$ be a function and $A' \subseteq A$ be a set. The function

$$\begin{aligned} f|_{A'} : A' &\to B, \\ x &\mapsto f(x) \end{aligned}$$

  is called the restriction of f to A'.
- Let $f \colon A \to B$ be a function and A' be a set with $A \subseteq A'$. The function $f' \colon A' \to B$ is called an extension of f, if the following property holds:

$$\forall a \in A \colon f'(a) = f(a)$$

Chapter 3

Writer Approach

A prerequisite of property-based testing is the generation of test cases. In a pure scenario, only the input of a function has to be generated. In the context of database interactions, it is often necessary to generate database content, as the behaviour of the function might depend on it. A straightforward approach is to first generate test data and manually insert the generated content into the database at the beginning of each test. While this is feasible in small scenarios, it can be very laborious, when there is a large quantity of relationships between different tables in the database. For the insertion of a single value into the database, it can be necessary to fill multiple other database tables beforehand, to avoid *referential integrity constraint* violations. Referential integrity constraints are constraints that enforce that every referenced value exists in the database.

We introduce an alternative approach, where both the generation and the insertion of the generated values into the database are encapsulated in a single monad called `GenWithDBActions`. This approach is very flexible, as these extended generators, which use the `GenWithDBActions` monad, can be defined similarly to normal generators. Additionally, the tester can define, which of the generated values are inserted into the database. Extended generators which were already defined can be reused in the definition of new generators, to avoid redundant code, which makes this approach very compositional.

3.1 Motivating Example

ScalaCheck uses the `Gen` monad to encapsulate the random generation of values. There already exist some predefined generators, e.g. one that generates an integer in the given range:

```
Gen.choose(min: Int, max: Int): Gen[Int]
```

For-comprehensions can be used to compose multiple generators into more complex generators. In the example from Section 2.2, a generator for users is implemented by combining three existing generators:

```
val userGen: Gen[User] = for {
    id <- Gen.uuid
    name <- Gen.alphaStr
    age <- Gen.choose(18, 80)
} yield User(id, name, age)
```

Generators can later be used to test whether certain properties are fulfilled for all generated values. This can be achieved using the `forAll` method.

```
def forAll[A](gen: Gen[A])(f: A => Prop): Prop
```

This method has two arguments: The first argument is the generator, and the second argument is a function, which checks whether a property is fulfilled for a generated value. A simple property can be defined as follows:

```
val prop = forAll(userGen) {
    user: User =>
        user.age <= 80 && user.age >= 18
  }
```

This test simply checks whether all generated users have an age in the correct interval. The property is defined with a Boolean expression, which is internally converted into a property of type `Prop`. The generator `userGen` is used to generate the user. This test is completely independent of the database. However, there are many scenarios, in which the tested function requires existing content in the database. Therefore, we need the functionality, which adds content into the database. Many database query libraries (e.g. Slick, Quill and Doobie) provide a monad, which encapsulates database actions. In this thesis, we will focus on the `DBIO` monad from the Slick library [Lightbend, 2012], but similar approaches are possible for other libraries. The type signature of a method that defines the database action to add a user can be defined as follows:

```
def insertUser(user: User): DBIO[User]
```

It is important to realize that calling this method does not have any effect on the database. Instead, the method only returns a `DBIO` value. This `DBIO` value can be executed at a later point to insert the value into the database. For this purpose, we use the method `run`, which takes a database action, executes it, and returns the result:

```
def run[A](dbio: DBIO[A]): A
```

In reality, `DBIO` actions are executed differently, but we stick to this method for simplicity. Now we can define a test, which inserts the generated user into the database.

```
val prop = forAll(userGen){
    user: User =>
        run(insertUser(user))

        //test something
}
```

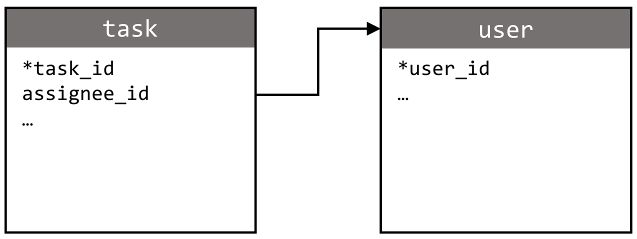

Figure 3.1: A database schema with users and tasks. Each task entry references an assigned user.

In this approach, all generated values are inserted into the database manually in each property. In small scenarios, as the example above, this is not a problem, but in more complex scenarios it can be a tedious and error-prone task, as the insertions have to be in the correct order.

Consider the database schema in Figure 3.1. There is a table for tasks and a table for users. Each task is assigned to a user. The generator of a `Task` takes the `UUID` of a user as an argument, as it must be referenced. As both the user and the task must be inserted into the database for the test, we need a combined generator. This generator creates both a task and a corresponding user.

```
case class Task(name: String, assigneeId: UUID)

def taskGen(assigneeId: UUID): Gen[Task] = ???

val setupGen: Gen[Setup] = for {
    user <- userGen
    task <- taskGen(assigneeId = user.id)
} yield Setup(user, task)
```

Note that both, the user and the task, are returned by the generator, even if the content of the user is irrelevant for the tested function. This is necessary, since the user must be inserted into the database in the property. Now a property can be defined:

```
val prop = forAll(setupGen){
    case Setup(user, task) =>
        run(insertUser(user))
        run(insertTask(task))

        //test something
}
```

With this approach, the user and the task have to be inserted manually. If the task had been inserted into the database before the user, a referential integrity constraint violation would occur, as the user does not exist yet. Therefore, the tester has to insert the values in the correct order, which is an error-prone effort. Clearly, this approach does not scale well, when database rows have numerous direct or indirect dependencies.

3.2 Generator Extension

To address the difficulties of manual database insertions, we introduce the monad `GenWithDBActions`, which encapsulates both the generation and the insertion of values. Intuitively, a value of type `GenWithDBActions[A]` can be viewed as a generator of type `Gen[(DBList, A)]`, which does not only generate a value, but a pair of the value and a list of database actions. For the list of database actions, we use the type `DBList`. The generators can be combined as previously, but for each generated value, a database action handling the insertion is appended to the `DBList`. After the generation is complete, all actions can be executed at once.

Instead of actually working on a pair, we use a concept called `Writer`, as directly generating a pair is impractical. A value `Writer[L, A]` of the `Writer` monad encapsulates a computation where a stream of data of type `L` is attached to the result of the computation of type `A`. This concept can be used, to collect logging entries during a computation [Grabmüller, 2006]. In this case, the type `L` is a list of logging entries. Another use case is code generation, where the code is successively appended to the result [Brown and Sampson, 2009, Axelsson, 2016].

For our purposes, we utilize a `Writer[DBList, A]`, which means that during the computation database actions are collected and attached to the result. As in our case the computation is a generator, this leads to the following type:

```
Gen[Writer[DBList, A]]
```

To simplify the access to the inner monad, we utilize the monad transformer `WriterT`. A monad transformer is a structure that combines the functionality of two nested monads into one [Liang et al., 1995]. In our case, the monad transformer combines the behaviour of the `Gen` monad and the `Writer` monad. The monad transformer does not change the semantic meaning but provides additional methods, which make the usage of the monads more practical. The first method useful to us is `liftF`:

```
def liftF[A](gen: Gen[A]): GenWithDBActions[A]
```

The `liftF` method exists in all monad transformers and converts (or *lifts*) a value of the outer monad into a value of the combined monad. This means that a normal generator `Gen[A]` can be converted into the extended generator `GenWithDBActions[A]`. The second method we use is `tell`:

```
def tell[A](action: DBIO[A]): GenWithDBActions[Unit]
```

The `tell` method is specific to the `Writer` monad and is used to attach a value to the stream of collected data of the computation. As we are collecting database actions, we use the `tell` method to attach a database action to the generator. The database action does not necessarily have to be an insertion, but insertions are the most common.

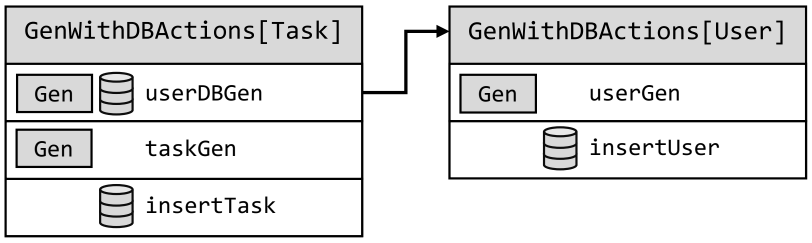

Figure 3.2: A visualization of the definition of `GenWithDBActions` generators. The generator for a task can be defined by using the generator for a user.

Now we can start using `GenWithDBActions` for our example. Similarly to normal generators, `GenWithDBActions` can be combined using for-comprehensions:

```
val userDBGen: GenWithDBActions[User] = for {
    user <- liftF(userGen)
    _ <- tell(insertUser(user))
} yield user
```

The first line of the for-comprehension represents the generation of a user, and the second line represents the insertion of the generated user into the database. When the `userDBGen` is defined, neither the user is generated nor is the user inserted into the database. Instead, the monadic value encapsulates the definition, how the user can be generated and inserted at a later point. This generator can be used in the definition of other `GenWithDBActions`, as we will show in the previous example from Figure 3.1:

```
val taskDBGen: GenWithDBActions[Task] = for {
    user <- userDBGen
    task <- liftF(taskGen(assigneeId = user.id))
    _ <- tell(insertTask(task))
} yield task
```

The definition of the `taskDBGen` is visualized in Figure 3.2. A few things are noteworthy in this example: Firstly, the user does not have to be inserted anymore, because the insertion is already encapsulated in `userDBGen`. Secondly, the `UUID` of the generated user can be used to define the generation of the task in the next line. This demonstrates the compositionality of this approach. And lastly, the return type `GenWithDBActions[Task]` does not have to contain the user anymore. This is because the insertion of the user is already encapsulated in the generator and therefore the user is no longer required. Using the `taskDBGen`, we can define the property again:

```
val prop = forAll(taskDBGen) {
    case ValueWithDBActions(task, actions) =>
        actions.foreach(run)

        //test something
}
```

Now the generation does not only return the task but both the task and the database actions. The type `ValueWithDBActions` is used as a container for the generated values and the database actions. Before implementing the actual test, we still have to run all database actions. However, in contrast to the previous approach, we do not have to manually execute all actions anymore but can execute all actions at once.

3.3 Refinements

As the pattern of executing all database actions in the beginning of a test occurs frequently, we implemented the method `forAllDB` in order to simplify the tests:

```
def forAllDB[A](
    genDB: GenWithDBActions[A])(
    f: A => Prop
): Prop = ???
```

The type signature is very similar to the `forAll` method. The main difference is that it takes an extended generator `GenWithDBActions[A]` instead of a regular generator `Gen[A]` as an argument. Internally, the `forAllDB` method executes all database actions and uses the `forAll` method to evaluate the property afterwards. Additionally, preliminary steps to prepare the database are included. Using this method, a property can be defined as follows:

```
val prop = forAllDB(taskDBGen) {
    task: Task =>

        //test something
}
```

It is noticeable that calling the `liftF` and `tell` methods consecutively is another frequently repeating pattern. This happens in situations when a given conventional generator `Gen` should be converted into a `GenWithDBActions` which only inserts the generated value into the database. An example is the user generator `userDBGen` we previously defined:

```
val userDBGen: GenWithDBActions[User] = for {
    user <- liftF(userGen)
    _ <- tell(insertUser(user))
} yield user
```

To simplify this pattern, we define an `createAndInsert` method:

```
def createAndInsert[A](
    gen: Gen[A])(
    implicit insertable: Insertable[A]
): GenWithDBActions[A]
```

In this method, the first parameter of the `createAndInsert` method is the generator. The second parameter is implicit, which means that the parameter does not

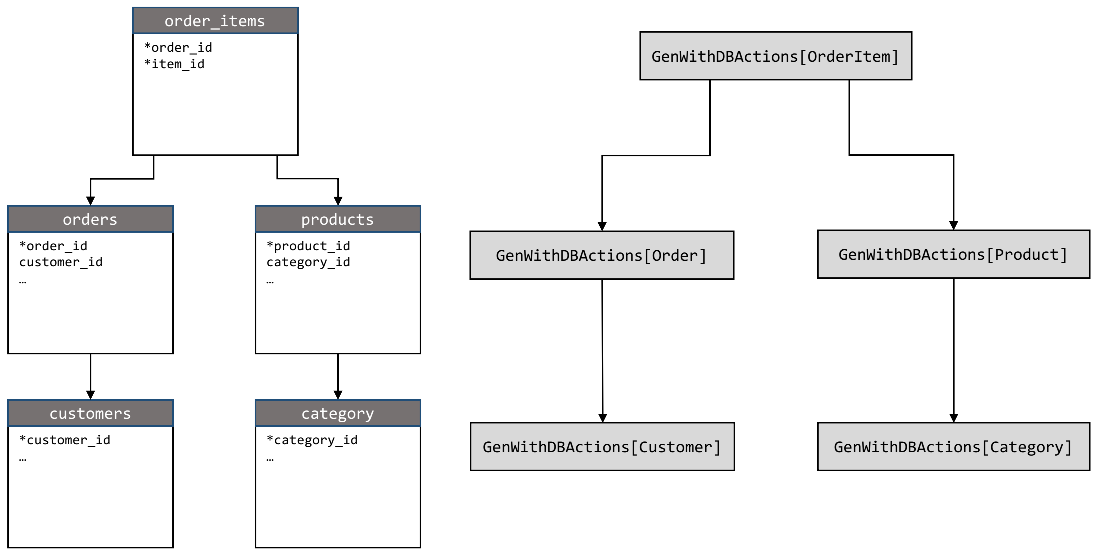

Figure 3.3: A demonstration of the compositionality of `GenWithDBActions`. When the database schema is hierarchically structured, the generators can be defined in the same structure.

have to be explicitly passed when the method is called. Instead, Scala searches in the current scope for a value with the specified type and automatically passes it as the argument. In this case, the implicit parameter is of type `Insertable[A]`. The type `Insertable[A]` encapsulates the information, how a value of type `A` can be inserted into the database. For the `createAndInsert` method to work, the tester has to provide an instance of `Insertable[A]` once. Given that this is the case, the definition of `userDBGen` can be simplified to the following expression:

```
val userDBGen: GenWithDBActions[User] = createAndInsert(userGen)
```

Using this method generally avoids the risk of forgetting to specify the database insertions in extended generators.

3.4 Discussion

The writer approach is especially suitable for hierarchically structured database schemas. An example of a hierarchically structured database schema is shown in Figure 3.3. In the example, customers can place orders. Each order contains a set of products. This relationship is represented by the `order_item` table. Additionally, products are assigned to product categories. To insert a single order item into the database it is necessary to insert an order, a customer, a product, and a product category beforehand. However, the extended generator for the order item does not use the generators for the customer and category directly, as the generators for the order and the product already include these generators, respectively. Therefore, it is not necessary to focus on transitive dependencies

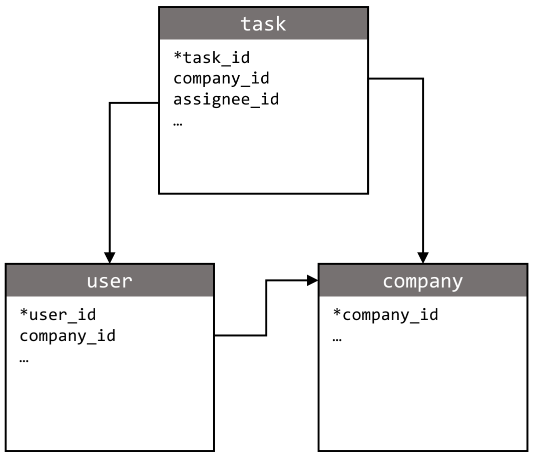

Figure 3.4: A database schema with tasks, users, and companies. Each task belongs to a company and has an assigned user. Additionally, the user references the company they work at.

in this approach. This makes the generators maintainable, as changes to their transitive dependencies do not affect the generators. Furthermore, redundant code is avoided by the compositionality of the generators.

The implementation of generators is not so straightforward when the database schema is not hierarchically structured. We show an example in Figure 3.4. In the example, tasks belong to a company and have an assigned user. The user references the company they work at. This database schema is not hierarchical, as there are two paths from a task to a company. It is possible to define the generator similarly to the previous example: Whenever a table references another table, the generator directly uses the generator of the other table. We visualize this generator structure in Figure 3.5. The generator of a task uses the generators of a user and a company. As the user belongs to a company, the user generator uses the generator of a company as well. A drawback of this approach is that it always generates the company of the user and the company of the task separately. Therefore, the case that the user and the task belong to the same company is never generated[1]. The scenario that the task and the user belong to the same company can be generated in the following manner: The generator of a task starts by generating a company. The ID of the generated company is passed to the user generator so that the generated user can reference the same company. In this case, the user and the company always belong to the same company. A generator that can generate both scenarios can be defined by randomly choosing between the two described generators. This example demonstrates that the writer approach is very flexible, but in some scenarios, manual effort is required

[1]This statement is only true under the assumption that the generator never returns the same company twice. This scenario cannot be generally ruled out. However, the probability is vanishingly small if UUIDs are used.

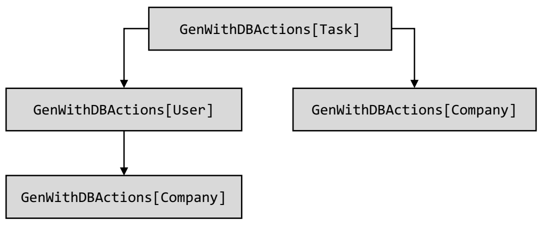

Figure 3.5: A possible way to implement generators for the database schema in Figure 3.4. In this case, the company of the user and the company of the task are always independent of each other. The scenario that the user and the task belong to the same company is never generated.

to avoid that corner cases are missed.

Chapter 4

Reference Graph Generator

When property-based tests are written for database applications, it is often sensible to insert data into the database as a preliminary step. For the insertion of rows into a database table, it is often necessary to insert rows into adjacent database tables beforehand, due to foreign key references. As the adjacent database tables can also have foreign key references, this can lead to long transitive reference chains. Depending on the database configuration, it might be necessary to add the rows in a specific order, to avoid integrity constraint violations. Doing this manually can be very laborious and cause a lot of boilerplate code. This can also be hard to maintain, as changes in the database schema can make a lot of changes in the test case generation necessary. Additionally, the generation of test cases must be implemented carefully, so rare corner cases are not omitted. We propose an algorithm to automate the process of creating references depending on the database schema. The implementation uses seed-based randomization such that the same seed always leads to the same generated graph.

4.1 Reference Schema

We focus only on the references between different rows and ignore the concrete content of the columns. Therefore, we speak of *proxies* instead of rows. To generate database content based on the proxies, the values of the columns need to be generated as well. This can be done after the generation of the references and is not a part of the proposed algorithm. We model both the schema and the content of the database similarly to heterogeneous information networks but utilize multigraphs.

Definition 4.1.1 (Multigraph). A (directed) multigraph is the structure $G = (V, E, s, t)$ where V is a finite set of vertices, E is a finite set of edges, $s\colon E \to V$ defines the source vertex of an edge and $t\colon E \to V$ defines the target vertex of an edge. Given an edge labelling function $f\colon E \to L_E$, then (V, E, s, t, f) is called an edge labelled multigraph. Given an additional vertex labelling function $g\colon V \to L_V$, then (V, E, s, t, f, g) is

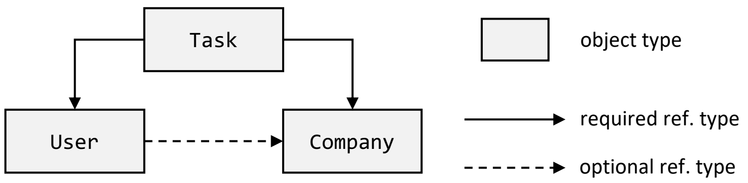

Figure 4.1: An example of a reference schema. There are tasks, users, and companies. A user can optionally work at a company. Each task belongs to a company and is assigned to a user.

called an edge and vertex labelled multigraph.

An alternative definition would represent a multigraph as a set of vertices and a multiset of edges. We choose this approach to be able to differentiate different edges with the same source and target vertex. The multigraph which represents the database schema is called the reference schema. We define a reference schema as follows:

Definition 4.1.2 (Reference Schema). A reference schema is an edge labelled multigraph $RS = (\mathcal{V}, \mathcal{E}, \mathit{sourceType}, \mathit{targetType}, \mathit{necessity})$, where $\mathit{necessity}\colon \mathcal{E} \to \{\ \mathit{required}, \mathit{optional}\ \}$ is the edge labelling function. We call the vertices *object types* and the edges *reference types*. For a reference type $rt \in \mathcal{E}$, we call $\mathit{sourceType}(rt)$ the source type, $\mathit{targetType}(rt)$ the target type. We say rt is required iff $\mathit{necessity}(rt) = \mathit{required}$, and rt is optional iff $\mathit{necessity}(rt) = \mathit{optional}$.

The vertices of a reference schema represent the database tables and the edges represent foreign key constraints. Thus, the semantics of a reference type between the object types v and w is that table v has a column that references the primary key column of the table w. We included the edge-labelling function *necessity* to be able to model NOT NULL constraints of foreign keys. The reference schema is a multigraph in order to allow multiple foreign key constraints between the same two tables. We do not model more complex database constraints. An example of a reference schema can be seen in Figure 4.1.

4.2 Reference Graph

Using the definition of a reference schema we can now define the multigraph, which represents the content of the database.

Definition 4.2.1 (Reference Graph). For a reference schema $RS = (\mathcal{V}, \mathcal{E}, \mathit{sourceType}, \mathit{targetType}, \mathit{necessity})$ an edge and vertex labelled and directed $RG = (V, E, s, t, \varphi, \psi)$ is called a *reference graph for RS* iff all the following statements hold:

1. $\varphi\colon V \to \mathcal{V}$

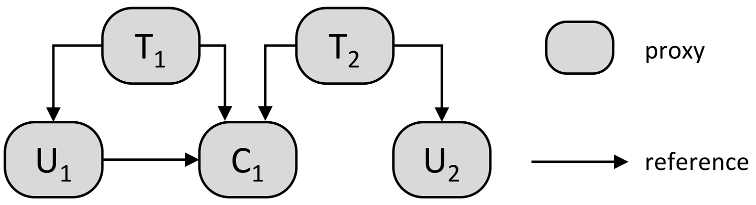

Figure 4.2: An example of a reference graph for the reference schema in Figure 4.1. C_1 is of type *Company*, U_1 and U_2 are of type *User* and T_1 and T_2 are of type *Task*.

2. $\psi\colon E \to \mathcal{E}$
3. $\forall e \in E\colon \mathit{sourceType}(\psi(e)) = \varphi(s(e)) \wedge \mathit{targetType}(\psi(e)) = \varphi(t(e))$.
4. $\forall v \in V\colon \forall rt \in \mathcal{E}\colon |\{\, e \in E \mid rt = \psi(e) \wedge s(e) = v \,\}| \leq 1$.

We call the vertices *proxies* and the edges *references*. For every $v \in V$ we call $\varphi(v)$ the type of v and for every $e \in E$ we call $\psi(e)$ the type of e. For a reference $e \in E$, we call $\mathit{sourceType}(\psi(e))$ the source type and $\mathit{targetType}(\psi(e))$ the target type of e. We call $d \in \mathcal{E}$ a dependency of $v \in V$ iff $\varphi(v) = \mathit{sourceType}(d)$ and we call it a backward dependency of $v \in V$ iff $\varphi(v) = \mathit{targetType}(d)$.

The vertices, or proxies, each represent a database row and the edges represent foreign key references. An example of a reference graph can be seen in Figure 4.2. The third property states that the types of the source and target of an edge match the definition of the reference schema. The fourth property states that for a proxy, there are never multiple references that satisfy the same constraint. However, by our definition, it is possible that for some required reference type there is no reference. This is intentional, since the property is not fulfilled in intermediate steps of the algorithm.

Definition 4.2.2 (Fully Defined). Let $RG = (V, E, s, t, \varphi, \psi)$ be a reference graph for the reference schema $RS = (\mathcal{V}, \mathcal{E}, \mathit{sourceType}, \mathit{targetType}, \mathit{necessity})$, $v \in V$ be a proxy and $rt \in \mathcal{E}$ be a reference type.

1. rt is satisfied for v $:\Longleftrightarrow \exists e \in E\colon \psi(e) = rt \wedge s(e) = v$
2. v is fully defined $:\Longleftrightarrow \forall rt \in \mathcal{E}\colon \mathit{necessity}(rt) = \mathit{required} \wedge \varphi(v) = \mathit{sourceType}(rt) \Rightarrow rt$ is satisfied for v.
3. RG is fully defined $:\Longleftrightarrow \forall v \in V\colon v$ is fully defined.

Therefore, a reference graph is fully defined, if every proxy has a reference for each required dependency of that proxy. We use the reference graph in Figure 4.2 as an example for the reference schema in Figure 4.1. Consider the reference type rt between the types *User* and *Company*. The reference type rt is satisfied for a proxy of type *User* if the proxy has an outgoing reference of this type. Since U_1 is a user and references the company C_1, *rt* is satisfied for U_1. However, *rt* is not satisfied for U_2, because U_2 has no corresponding reference. A proxy is fully defined if all required dependencies are satisfied. The proxies U_1 and U_2 are both fully defined, since their only dependency *rt* is optional. The reference graph in Figure 4.2 is fully defined, as all its proxies are fully defined.

The content of reference graphs should be inserted into the database at some point. Depending on the configuration of the database, it can be non-trivial to decide in which order the rows should be added to the database, as an arbitrary order can lead to referential integrity constraint violations. When a proxy v references another proxy w, then the corresponding database row of v cannot be inserted into the database before w. The concept of topological orderings can be useful to address this challenge.

Definition 4.2.3 (Topological Ordering). Let $G = (V, E, s, t)$ be a multigraph. The list $v \in V^{|V|}$ is called a topological ordering if and only if the following properties hold:

1. $\forall i, j \in \{1, \ldots, |V|\} : i \neq j \implies v_i \neq v_j$

2. $\forall e \in E \colon \forall i, j \in \{1, \ldots, |V|\} : s(e) = v_i \wedge t(e) = v_j \implies i < j$

G is called acyclic iff a topological ordering exists for G.

When a topological ordering of the reference graph is known, the rows can be inserted into the database in the reversed order. We designed the algorithm to return not only a reference graph but also a topological ordering. It is a known property of graphs that a topological ordering of a graph only exists if and only if the graph has no directed cycles. For simplicity, we defined the term *acyclic* with the existence of a topological ordering instead of the absence of a cycle. A consequence of the *acyclic* property is that a topological ordering does not exist for some reference graphs. Therefore, we decide to restrict the generated reference graphs to acyclic reference graphs.

4.3 The Algorithm

The algorithm we propose can be seen in Algorithm 1. The input is an arbitrary reference schema $RS = (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$, a set of predefined initial proxies *IP* and an amount of iterations. All initial proxies must have a type from $\mathcal{V}$ assigned. The output is a reference graph for *RS*, which contains all the predefined proxies *IP*. For a proxy $v \in V$ we call a reference $r \in E$ a *forward reference* of v iff $s(r) = v$ and a *backward reference* iff $t(r) = v$. The algorithm uses a method *ExpandForward*, which visits proxies to generate forward references, and a method *ExpandBackward*, which visits proxies to generate backward references. The algorithm proceeds by alternatingly calling the

```
Input: The reference schema $RS$,
set of initial proxies $IP$,
$iteration\_amount \in \mathbb{N}$
Output: The reference graph and a topological ordering of the proxies.
$RG$ = initial reference graph with $V = IP$ and $E = \emptyset$;
mark all vertices as unvisited;
$RG, partial\_ordering = ExpandForward\,(RS, RG)$;
$ordering = partial\_ordering$;
repeat $iteration\_amount$ times
    $RG = ExpandBackward\,(RS, RG)$;
    $RG, partial\_ordering = ExpandForward\,(RS, RG)$;
    $ordering = partial\_ordering$ ++ $ordering$;
end
return $RG$, $ordering$;
```

Algorithm 1: The proposed algorithm to generate a random reference graph based on a reference schema.

ExpandForward and *ExpandBackward* methods. Calling the *ExpandForward* and *ExpandBackward* methods only once would restrict the set of reference graphs that can be generated. After a set amount of iterations, which is defined by *iteration_amount*, the algorithm terminates. The algorithm marks the proxies mutually exclusively, depending on which references were already added:

unvisited The proxy has not been visited yet. It is possible that the proxy was newly created by the algorithm or was one of the predefined proxies.

forward_visited The proxy has been visited once to create forward references and is fully defined. However, the algorithm did not visit the proxy a second time yet, to generate backwards references.

backward_visited The proxy has been visited twice: Once to generate forward references and once to generate backward references.

Each time the *ExpandForward* method was executed, the reference graph only consists of *forward_visited* and *backward_visited* proxies. Each time the *Expand-Backward* method was executed, the reference graph only consists of proxies marked as *unvisited* or *backward_visited*. The algorithm calls the *ExpandForward* method last, to return a fully defined reference graph. For simplicity, we do not explicitly define the functions φ and ψ of the reference graph but define the type of each proxy and reference once it is generated.

```
Function ExpandForward:
    Input: The reference schema RS,
    the current reference graph RG
    Output: The updated reference graph and an ordering of the visited
            proxies
    ordering = empty list;
    while there are unvisited proxies in RG do
        selected = randomly pick unvisited proxy from RG;
        for dependency ← all non-satisfied dependencies of selected do
            /* Boolean decides whether reference is generated
               for optional dependency                              */
            doesGenerateOptional = randomly generate Boolean;
            if (dependency is required) or doesGenerateOptional then
                target_type = the target type of dependency;
                /* set of potential targets of new edge             */
                POT =
                  $\{ p \mid (p \text{ is not } \mathit{forward\_visited}) \land \varphi(p) = \mathit{target\_type} \} \setminus$
                  $\{ \mathit{selected} \}$ ;
                /* Boolean decides whether existing proxy or
                   newly created proxy is target                    */
                useExistingProxy = randomly generate Boolean;
                if ($POT \neq \emptyset$) and useExistingProxy then
                    target = pick random element of POT;
                else
                    target = new proxy of type target_type;
                    add proxy target marked as unvisited to RG;
                end
                add edge from selected to target of type dependency to RG;
            end
        end
        mark selected as forward_visited;
        append selected to ordering;
    end
    return RG, ordering
end
```

Algorithm 2: The method to create forward references.

4.4 Expand Forward

The *ExpandForward* method, which is responsible to generate all required forward references, can be seen in Algorithm 2. It proceeds by successively visiting *unvisited* proxies in a random order and generating forward references for the selected proxy. For all required dependencies of the selected proxy, the necessary references are generated, so it is fully defined afterwards. The proxy which is being referenced can either be a new proxy or an existing proxy of the correct type. The random Boolean *useExistingProxy* determines which of these cases is chosen. However, a new proxy is always generated, if no suitable existing proxy exists. It is also decided randomly, whether a reference is created for an optional dependency. This random decision is determined by the random Boolean *doesGenerateOptional*. Note that the set *POT* does not contain the proxies marked as *forward_visited*, so no references to proxies which were visited precisely once are generated. This is intentional, as due to this decision, only acyclic reference graphs are generated. It is later shown that, apart from acyclicity, this decision does not further restrict the set of reference graphs that can be generated. The list *ordering* is the order in which the proxies were visited. It is later helpful to find a topological ordering of the proxies but has no further purpose.

4.5 Expand Backward

The *ExpandBackward* method, which is responsible for adding backward references, is shown in Algorithm 3. Every proxy marked as *forward_visited* is visited and a random amount of backward references is added. The source of the reference is always a new proxy of the corresponding type. The generation of forward references and backward references was separated, as otherwise, depending on the exact definition, cycles could be generated.

4.6 Example

As an example, we are going to demonstrate, how the reference graph in Figure 4.2 can be generated using Algorithm 1. The execution of the algorithm in this example is visualized in Figure 4.3. We assume that T_1 is the only initial proxy. The algorithm starts by calling ExpandForward. T_1 is of type *Task*, which means that there are two required dependencies of T_1: One to *User* and one to *Company*. Neither a user nor a company exists yet. Therefore, when T_1 is visited by ExpandForward in step 2, a proxy of type *Company* and a proxy of type *User* must be generated. We refer to the generated user as U_1 and the generated company as C_1. As both U_1 and C_1 are marked as *unvisited*, the algorithm decides randomly which of the proxies to visit in step 3.

```
Function ExpandBackward:
    Input: The reference schema $RS$,
    the current reference graph $RG$
    Output: The updated reference graph
    for $selected \leftarrow$ all proxies marked as forward_visited do
        mark $selected$ as $backward\_visited$;
        for $backwardDependency \leftarrow$ all backward dependencies of selected do
            $amount$ = randomly choose one of $\mathbb{N}$;
            $source\_type = sourceType(backwardDependency)$;
            repeat $amount$ times
                $source$ = new proxy of type $source\_type$;
                add proxy $source$ marked as $unvisited$ to $RG$;
                add edge from $source$ to $selected$ of type
                  $backwardDependency$ to $RG$;
            end
        end
    end
    return $RG$
end
```

Algorithm 3: The method to create backward references.

We assume that U_1 is visited first. As U_1 is of type *User*, there is only one optional dependency to *Company*. There are three different possible outcomes:

1. The algorithm does not add a dependency, as the only dependency is optional. This is the case if *doesGenerateOptional* is *false*.

2. Since C_1 is unvisited and of type *Company*, the algorithm generates a reference from U_1 to C_1.

3. The algorithm generates a new proxy of type *Company*. This only happens when *doesGenerateOptional* is *true* and *useExistingProxy* is *false*.

In our case, the algorithm chose the second option and a reference from U_1 to C_1 is generated. In step 4, the algorithm can only visit C_1, as it is the only unvisited proxy. Since C_1 has no dependencies, no references can be generated and ExpandForward terminates. The reference graph is fully defined.

If and how often ExpandBackward is called depends on the *iteration_amount* parameter. We assume that *iteration_amount* is set to one and ExpandBackward is executed a single time. T_1 does not have any backward dependencies, because no object type references tasks in the reference schema in Figure 4.1. Therefore, no backward references can be generated for T_1 in step 5. In step 6, U_1 is visited. The only backward dependency of U_1 is to *Task*, which means that a random amount of tasks, which reference U_1, is generated. In our case, the random

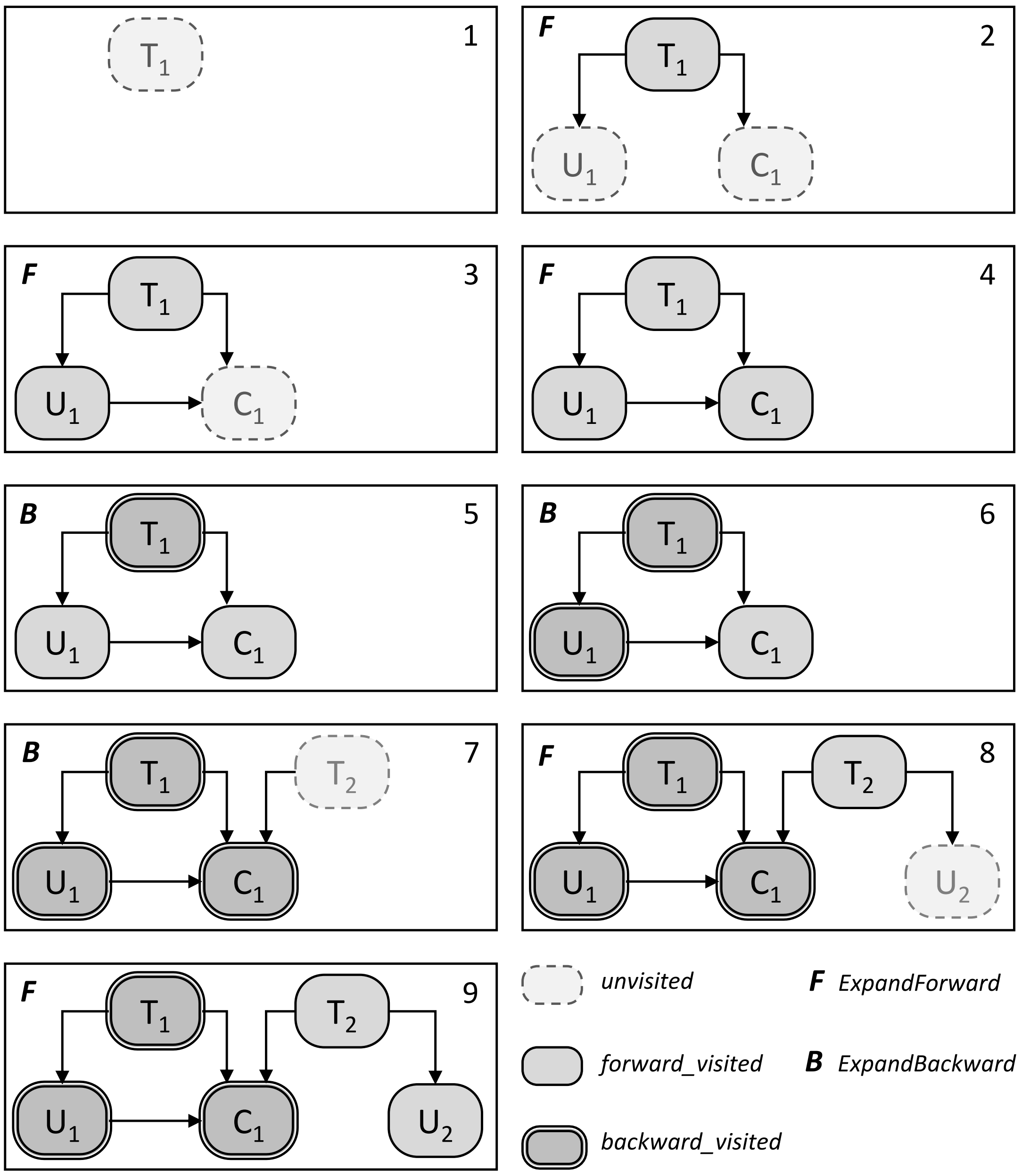

Figure 4.3: An example execution of Algorithm 1, which demonstrates how the reference graph in Figure 4.2 can be generated. There are two executions of the ExpandForward method and one execution of the ExpandBackward method.

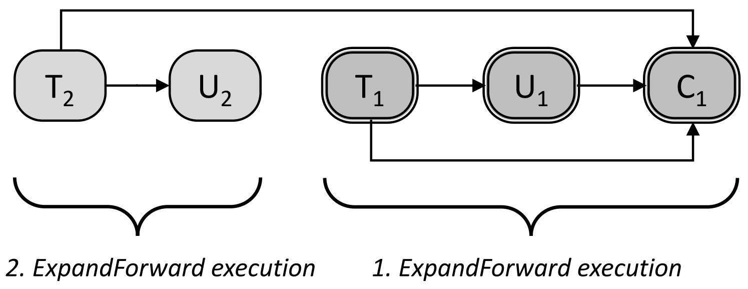

Figure 4.4: The returned ordering of the example execution in Figure 4.3. The proxies are topologically ordered, as all proxies only reference later proxies.

amount is 0 and no further tasks are generated. C_1 has two backward dependencies: *Task* and *User*. Therefore, a random amount of tasks and users, which reference C_1, is generated in step 7. In this case, this leads to the generation of the proxy T_2 of type *Task*. As no further proxies are marked as *forward_visited*, ExpandBackward terminates.

At this point, the reference graph is not fully defined, as T_2 does not reference a user. In order to return a fully defined reference graph, the algorithm generally executes ExpandForward after every ExpandBackward execution. This ExpandForward execution is shown in the steps 8 and 9. Afterwards, the algorithm returns the generated reference graph.

We show the ordering returned in this example in Figure 4.4. The proxies are topologically ordered, as all proxies only reference later proxies. The proxies of one ExpandForward execution are in the order in which they were visited: In the first ExpandForward execution, the proxies were visited in the order T_1, U_1, C_1. The proxies of the second ExpandForward execution were visited in the order T_2, U_2. But the proxies of the first ExpandForward execution are listed after the proxies of the second ExpandForward execution. This is necessary as proxies from the second ExpandForward execution can reference proxies from the first ExpandForward execution.

4.7 Properties of the Algorithm

In this section, we prove two properties of the algorithm:

Correctness All generated reference graphs fulfil certain properties, e.g. they are fully defined.

Completeness All relevant reference graphs can be generated by the algorithm and no corner cases are missed.

The completeness is important because an incomplete generator would lead to an incomplete test. The descriptions of correctness and completeness are vague, as we do not describe, which reference graphs are *relevant* and what is meant by *certain properties*. In order to avoid ambiguity, we introduce a term called *desired*. We prove that all generated reference graphs are desired, and vice versa, all desired reference graphs can be generated by the algorithm. This implies that the set of all reference graphs that can be generated is equal to the set of all desired reference graphs. To define the term desired, we first need to define undirected paths:

Definition 4.7.1 (Undirected Path). Let $G = (V, E, s, t)$ be a multigraph, $n \in \mathbb{N}$, $v_0, \ldots, v_n \in V$, and $d_1, \ldots, d_n \in \{ \triangleright, \triangleleft \}$. An undirected path is the pair $p = (v_0, ((v_i, d_i))_{i=1}^{n})$ with the following properties:

1. $\forall i \in \{ 1, \ldots, n \} : (d_i = \triangleright) \Rightarrow (\exists e \in E \colon s(e) = v_{i-1} \wedge t(e) = v_i)$
2. $\forall i \in \{ 1, \ldots, n \} : (d_i = \triangleleft) \Rightarrow (\exists e \in E \colon t(e) = v_{i-1} \wedge s(e) = v_i)$

We define $\mathit{source}(p) := v_0$ and $\mathit{target}(p) := v_n$. We call $\triangleright$ a forward step and $\triangleleft$ a backward step.

For simplicity, we relax the notation in some occasions and write $(v_0\ d_1\ v_1 \ldots d_n\ v_n)$ instead of $p = (v_0, ((v_i, d_i))_{i=1}^{n})$. Therefore, the path $(v_0, ((v_1, \triangleleft), (v_2, \triangleright)))$ is written as $(v_0 \triangleleft v_1 \triangleright v_2)$. The definition of desired consists of five different properties:

Definition 4.7.2 (Desired). Let *IP* be a set of initial proxies, $RS = (\mathcal{V}, \mathcal{E}$, *sourceType, targetType, necessity*$)$ be a reference schema, and $\varphi_0 : IP \to \mathcal{V}$ be a function, which assigns types to the proxies in *IP*. Let $RG = (V, E, s, t, \varphi, \psi)$ be a reference graph for *RS*. *RG* is called desired if and only if all the following properties are fulfilled:

1. *RG* is acyclic.
2. *RG* is fully defined.
3. For every proxy $v \in V$ there exists an undirected path p with $\mathit{source}(p) \in IP$ and $\mathit{target}(p) = v$.
4. $IP \subseteq V$.
5. φ is an extension of φ_0. This means that $\forall v \in IP \colon \varphi(v) = \varphi_0(v)$.

The first property is necessary for a topological ordering to exist. Also, in many domains, cyclic data should be avoided. This restriction is further discussed in Section 4.8.2. The reference graph needs to be fully defined, as otherwise NOT NULL constraints in the database would be violated. The third property states that all proxies that are generated are connected to at least one of the initial proxies in *IP*. This avoids that the algorithm generates data that is irrelevant for the test. This restriction is also discussed in Section 4.8.2. The last two properties state that the algorithm does not change the type or remove the initial proxies in *IP*.

In Section 4.7.1 we prove that all reference graphs generated by the algorithm are desired. We continue by proving that all desired reference graphs can be generated by the algorithm in Section 4.7.2.

4.7.1 Correctness

In this subsection, we prove that all generated reference graphs are desired. As the definition of desired consists of five different properties, we prove each of the properties separately. We start with the property that the generated reference graph contains all initial proxies.

Lemma 4.7.1 (Initial Proxies are Contained) *Let IP be the initial proxies of the input and $RG = (V, E, s, t, \varphi, \psi)$ be the returned reference graph from Algorithm 1. Then, $IP \subseteq V$ holds.*

Proof. The algorithm starts by adding all proxies from *IP* to the reference graph. The algorithm never removes any proxies. Therefore, $IP \subseteq V$ holds. □

The next property states that all proxies are generated for a reason, i.e. for all generated proxies there is an undirected path from one of the initial proxies to the generated proxy.

Lemma 4.7.2 (Is IP Connected) *Let IP be the initial proxies of the input and $RG = (V, E, s, t, \varphi, \psi)$ be the returned reference graph from Algorithm 1. Then, for every proxy $v \in V$ exists an undirected path p with $source(p) \in IP$ and $target(p) = v$.*

The proof for this theorem is omitted. It can be shown by an induction on the edges in the order in which they were added. Initially, all proxies have such a path, as $V = IP$ and empty paths are allowed. Whenever a new edge is added, either the source or the target must have existed previously. Therefore, the path to the previously existing proxy can be extended to the newly created proxy and the statement still holds.

For the proofs that all generated reference graphs are fully defined and acyclic, we need an auxiliary loop invariant. The loop invariant states that no proxies can be unvisited between iterations of the *while* loop.

Lemma 4.7.3 (Loop Invariant) *Before and after each iteration of the while loop of Algorithm 1, each proxy is either marked as forward_visited or backward_visited.*

Proof. Whenever a proxy is added to the reference graph, the algorithm marks it. So every proxy is marked in the algorithm. The ExpandForward method only terminates, when no proxy is marked as *unvisited*. Before the first iteration of the *while* loop, the method ExpandForward was once called. Before each new iteration of the *while* loop, ExpandForward was called at the end of the previous iteration of the *while* loop. This means that at the beginning of each iteration, no proxy is marked as *unvisited*. As every proxy has to be marked, it is an invariant of the *while* loop that each proxy is either marked as *forward_visited* or *backward_visited*. □

Using the loop invariant, we can now prove that all generated reference graphs are fully defined.

Lemma 4.7.4 (Is Fully Defined) *Let $RG = (V, E, s, t, \varphi, \psi)$ be the reference graph of the result of an execution of the algorithm. Then, RG is fully defined.*

Proof. The reference graph is fully defined iff all proxies are fully defined. Let $v \in V$ be a proxy. We know from Lemma 4.7.3 that v is either marked as *forward_visited* or *backward_visited*. In both cases, v has been visited in ExpandForward. Consider the step, in which v was visited. Let $d \in \mathcal{E}$ be a required dependency of v. We have to show that d is satisfied for v after v was visited. If d was already satisfied previously, it is also satisfied afterwards, as the algorithm does not remove edges. If d was not satisfied previously, there is an iteration of the *for* loop in which d is selected, as all non-satisfied dependencies of v are chosen. Because d is required, the outer *if* block is executed. This implies that an edge from v with type d is created. Therefore, d is satisfied, and v is fully defined. □

To show that all generated reference graphs are acyclic, we use the ordering of the proxies, which is returned by the algorithm. We prove that this ordering is a topological ordering of all proxies of the reference graphs. This implies that the reference graph is acyclic, since it is a known property of graphs that topological orderings only exist for acyclic graphs. The returned ordering of the proxies, which we use, is composed of partial orderings, which are returned from ExpandForward. We start by proving that these partial orderings are topologically ordered.

Lemma 4.7.5 (ExpandForward Ordering) *Let $RG = (V, E, s, t, \varphi, \psi)$ be a reference graph and $(v_1, \ldots, v_n) \in V^n$ such that $(RG, (v_1, \ldots, v_n))$ is the result of an execution of the ExpandForward method. Then*

$$\forall i, j \in \{1, \ldots, n\}, e \in E : s(e) = v_i \wedge t(e) = v_j \implies i < j$$

holds.

Proof. Let $i, j \in \{1, \ldots, n\}$ and $e \in E$ with $s(e) = v_i$ and $t(e) = v_j$. Suppose, for the sake of contradiction, that $i \geq j$.

Both v_i and v_j were visited in the current ExpandForward execution, as they were added to the *ordering* list. We know that $v_i = v_j$ is impossible, as there is an edge between these proxies and the algorithm does not create self-references. Therefore, $i > j$ holds. As the algorithm only adds elements to the *ordering* list, when they are visited, we know that v_i was visited after v_j.

Case 1: The edge e was created in the current ExpandForward execution.

This implies that e was created when v_i was visited, as $s(e) = v_i$. We now consider the step, where e is created. As v_j has been visited before v_i, v_j must be marked as *forward_visited*. As $t(e) = v_j$, $target = v_j$ holds. Therefore, either $v_j \in POT$ holds or v_j is newly created. The former is impossible as v_j is marked as *forward_visited* and *POT* only contains proxies marked as *unvisited* or *backward_visited*. The latter is impossible, as v_j has already been visited before. This is a contradiction.

Case 2: The edge e already existed before the current ExpandForward execution.

Then, v_i and v_j also already existed before this ExpandForward execution. As v_i and v_j were visited in this execution, they were marked as *unvisited*. However, the edge e is not possible, when both v_i and v_j have not been visited before. This is a contradiction.

□

To complete the acyclicity proof, we need another invariant of the algorithm. This invariant states that once a proxy was visited, the algorithm cannot generate an outgoing reference for this proxy at a later point.

Lemma 4.7.6 (Completed Proxies) *Consider any step of the algorithm. Let v be a proxy marked as forward_visited or backward_visited. There is no later step of the algorithm where an edge e with $s(e) = v$ is created.*

Proof. We only need to look at the case that v is marked as *forward_visited*, since every proxy marked as *backward_visited* was marked as *forward_visited* previously. Therefore, proving the statement for all proxies marked as *forward_visited*, also proves the statement for all proxies marked as *backward_visited*. Let's assume, for the sake of contradiction, that there is a later step of the algorithm, where an edge e with $s(e) = v$ is created.

Case 1: The edge e is created in ExpandForward.

As $s(e) = v$, the edge e is created when v is visited. However, v already has been visited previously, as v was marked as *forward_visited*. This is a contradiction.

Case 2: The edge e is created in ExpandBackward.

As $s(e) = v$, v must be a newly created proxy. However, v already existed previously, as v is marked as *forward_visited*. This is a contradiction.

□

Finally, we prove that the returned ordering of the algorithm is a topological ordering, and the reference graph is acyclic. We specify a loop invariant, which states that the ordering is topologically ordered before and after each iteration of the *while* loop. This implies that the ordering is also topologically ordered when it is returned. Additionally, we have to show that all proxies uniquely occur in this ordering.

Theorem 4.7.1 (Topological Ordering) *Let $(v_1, \dots, v_n)$ be the returned ordering and $RG = (V, E, s, t, \varphi, \psi)$ be the returned reference graph of the algorithm. Then, $(v_1, \dots, v_n)$ is a topological ordering of RG and RG is therefore acyclic.*

Proof. To prove that $(v_1, \dots, v_n)$ is a topological ordering of RG, we need to show three properties:

1. $(v_1, \dots, v_n) \in V^{|V|}$
2. $\forall i, j \in \{ 1, \dots, |V| \} : i \neq j \Longrightarrow v_i \neq v_j$
3. $\forall e \in E \colon \forall i, j \in \{ 1, \dots, |V| \} : s(e) = v_i \wedge t(e) = v_j \Longrightarrow i < j$

We know from Lemma 4.7.3 that each proxy is either marked as *forward_-visited* or *backward_visited* when the algorithm terminates. Every proxy marked as *backward_visited* was marked as *forward_visited* previously Therefore, ExpandForward visited all proxies and added them to *ordering*. This implies that every proxy in V must appear in $(v_1, \dots, v_n)$. The list *ordering* cannot contain duplicates, as only *unvisited* proxies are added. This is already sufficient to see that the first two properties are fulfilled. To prove the last property, we construct a loop invariant proof.

Loop Invariant: Let $RG = (V, E, s, t, \varphi, \psi)$ be the current reference graph and $(v_1, \dots, v_n)$ be the current *ordering* before an iteration of the *while* loop. Then

$$\forall i, j \in \{ 1, \dots, n \}, e \in E : s(e) = v_i \wedge t(e) = v_j \Longrightarrow i < j$$

holds.

Initialization: RG and $(v_1, \dots, v_n)$ are the result of the first ExpandForward execution. Due to Lemma 4.7.5, this implies that the invariant is satisfied.

Maintenance: We now look at an arbitrary iteration of the *while* loop. Let $(v_1, \dots, v_m)$ be the partial ordering returned from ExpandForward and $RG' = (V', E', s', t', \varphi', \psi')$ be the updated reference graph returned from ExpandForward. We know from Lemma 4.7.5 that

$$\forall i, j \in \{ 1, \dots, m \}, e \in E' : s'(e) = v_i \wedge t'(e) = v_j \Longrightarrow i < j. \tag{4.1}$$

Let $RG = (V, E, s, t, \varphi, \psi)$ be the reference graph and $(v_{m+1}, \dots . v_{m+n})$ be the *ordering* before the current iteration of the *while* loop. We assume that the loop invariant is fulfilled for RG and $(v_{m+1}, \dots, v_{m+n})$:

$$\forall i, j \in \{ m+1, \dots, m+n \}, e \in E : s(e) = v_i \wedge t(e) = v_j \implies i < j. \tag{4.2}$$

We need to show that the loop invariant is fulfilled for the reference graph RG' and the combined ordering

$$(v_1, \dots, v_m, v_{m+1}, \dots, v_{m+n}).$$

Let

$$i, j \in \{ 1, \dots, m, m+1, \dots, m+n \}$$

and $e \in E$ be an edge with $s'(e) = v_i$ and $t'(e) = v_j$.

Case 1: $i, j \in \{ 1, \dots, m \}$.

We know from Equation 4.1 that $i < j$.

Case 2: $i, j \in \{ m+1, \dots, m+n \}$.

Therefore, $v_i, v_j \in V$ existed before the current *while* iteration. That means that v_i and v_j are marked as *forward_visited* or *backward_visited*, due to Lemma 4.7.3. If $e \notin E$ holds, then e must have been created in the current *while* iteration. This is not possible due to Lemma 4.7.6. If $e \in E$ holds, then we know $i < j$ from Equation 4.2.

Case 3: $i \in \{ 1, \dots, m \}, j \in \{ m+1, \dots, m+n \}$. Then, the following sequence of inequalities holds:

$$\begin{aligned} i &\leq m && i \in \{ 1, \dots, m \} \\ &< m+1 && \\ &\leq j && j \in \{ m+1, \dots, m+n \} \end{aligned}$$

Case 4: $i \in \{ m+1, \dots, m+n \}, j \in \{ 1, \dots, m \}$.

Then, v_i was already visited before the current iteration of the *while* loop and v_j was not visited in a previous iteration. Lemma 4.7.3 implies that v_j must have been created in the current iteration. Therefore, e must have been created in the current iteration of the *while* loop as well. This is not possible due to Lemma 4.7.6.

□

In the following theorem, we conclude the correctness proof by showing that all generated reference graphs are desired. We state that φ_0 is an input of Algorithm 1, although φ_0 is not mentioned in the pseudocode. The reason is that, for simplicity, the explicit definitions of the type functions are omitted in the algorithm. Instead, the pseudocode defines the type, whenever a reference or a proxy is generated. However, to express the following theorem formally, it is necessary to define the types of the proxies in *IP*. This is achieved using the function φ_0.

Theorem 4.7.2 (Correctness) *Let* $RS = (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$ *be a reference schema, IP be a set of initial proxies and* $\varphi_0 : IP \to \mathcal{V}$ *be a function, which assigns types to the proxies in IP. Suppose Algorithm 1 is executed with RS, IP, and* φ_0 *as the input and* $RG = (V, E, s, t, \varphi, \psi)$ *is the output. Then, RG is desired.*

Proof. To show that *RG* is desired, we need to show the following properties:

1. *RG* is acyclic.
2. *RG* is fully defined.
3. For every proxy $v \in V$ there exists an undirected path p with $source(p) \in IP$ and $target(p) = v$.
4. $IP \subseteq V$.
5. φ is an extension of φ_0. This means that $\forall v \in IP\colon \varphi(v) = \varphi_0(v)$.

We know from Theorem 4.7.1 that *RG* is acyclic. Lemma 4.7.4 states that *RG* is fully defined. The third property is fulfilled due to Lemma 4.7.2. In Lemma 4.7.1 it was stated that $IP \subseteq V$.

The last property is the only property that has not been proven yet. Let $v \in IP$. We need to prove that $\varphi(v) = \varphi_0(v)$. $\varphi_0(v)$ is the type of v in the input of the algorithm. $\varphi(v)$ is the type of v in the reference graph of the output. The type functions are not explicitly defined in the pseudocode, but at no point of the algorithm is the type of a proxy changed. Therefore, $\varphi(v) = \varphi_0(v)$ holds. □

4.7.2 Completeness

The next thing we prove is that, given a reference schema and the initial proxies, every desired reference graph can be generated by the algorithm. This means that a set of random choices exists, such that the algorithm outputs the specified reference graph. We prove the statement by constructing such an execution. In the construction, we define how the algorithm should behave in all random steps. Instead of being randomized, the algorithm can be viewed as being deterministic, where all random choices are controlled externally.

We start by classifying the proxies of the reference graph. The method ExpandForward creates forward references, i.e. if an edge $e \in E$ is generated, $s(e)$ exists already. The method ExpandBackward only creates backward references, i.e. $t(e)$ exists already. When a reference $e \in E$ is generated in ExpandBackward,

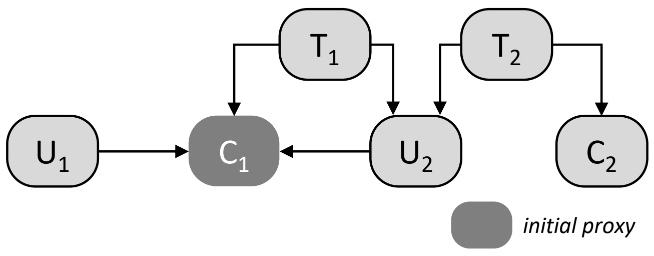

Figure 4.5: An example of a reference graph for the reference schema in Figure 4.1. *C_1* is the only proxy from the initial proxy set *IP*.

then $s(e)$ is always a newly created proxy marked as *unvisited*. Therefore, $s(e)$ cannot be visited in the same ExpandBackward execution. This means that only one layer of new proxies is generated, i.e. a direct predecessor of an existing vertex can be generated, but not a transitive predecessor. ExpandBackward has to be called multiple times, to generate a transitive predecessor. As an example, consider the reference graph in Figure 4.5. Assuming that U_2 is generated in a given ExpandBackward execution, then T_2 cannot be generated in the same execution of ExpandBackward, because U_2 is still marked as *unvisited*. However, in the next iteration of ExpandBackward, T_2 can be generated as well. Therefore, we separate the proxies into different layers, such that all proxies in the same layer can be generated in the same iteration of the algorithm. To achieve this, we introduce a concept called *rank*, which labels proxies w.r.t. the layer in which they are located:

> **Definition 4.7.3 (Rank).** Let $RS = (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$ be a reference schema and *IP* be a set of initial proxies. Let $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph. Let $p = (v_0, ((v_i, d_i))_{i=1}^{n})$ be an undirected path. We call p an *IP-path* of $v \in V$ iff $v_0 \in IP$ and $v_n = v$. We know that for all $v \in V$ an IP-Path must exist, due to the third property of *desired*. An IP-path of v is minimal iff no IP-path of v with fewer backward steps exists. The $rank_{RG}(v)$ is defined as the amount of backward steps of a minimal IP-path. When the reference graph is unambiguous, we also write $rank(v)$. Let $n \in \mathbb{N}$. The proxy v is called an n-rank-source of *RG* if and only if the following properties are fulfilled:
>
> 1. $rank_{RG}(v) = n$.
> 2. $\forall w \in V\colon (rank_{RG}(w) = n \implies \neg\exists e \in E\colon s(e) = w \wedge t(e) = v)$.

For some $n \in \mathbb{N}$ a proxy is an n-rank-source iff the proxy is not referenced by another proxy of the same rank. This can be visualized as a source node in the subgraph which only contains proxies of the same rank. Figure 4.6 shows the rank of the proxies from the reference graph in Figure 4.5. The proxy C_1 has a rank of 0, because it is an initial proxy itself and therefore the minimal IP-path

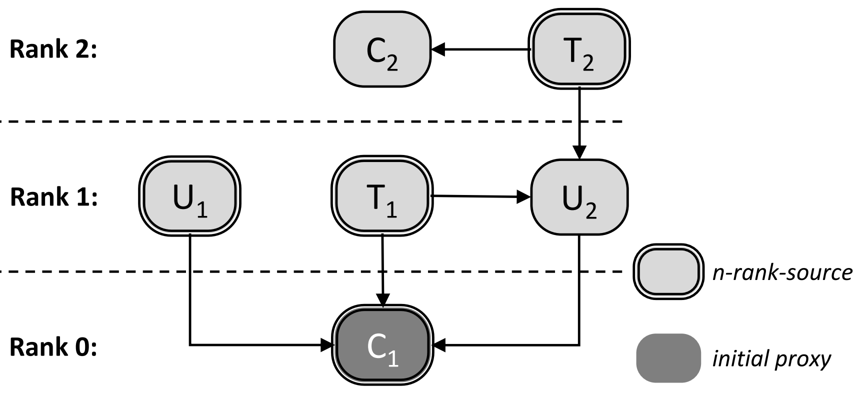

Figure 4.6: An overview of the rank of the proxies from the reference graph in Figure 4.5.

	Condition	Proxy generated in
$rank(v) = 0$	$v \in IP$	*Not generated*
	$v \notin IP$	First ExpandForward
$rank(v) = n$ with $n \in \mathbb{N}_{\geq 1}$	v is an n-rank-source	ExpandBackward in n-th iteration
	v is not an n-rank-source	ExpandForward in n-th iteration

Table 4.1: An overview of the part of the algorithm in which each proxy is generated in the execution we construct.

is (C_1). The minimal IP-path of the proxy U_1 is $(C_1 \lhd U_1)$, which leads to a rank of 1. Additionally, U_1 is a 1-rank-source, because it is not referenced by a proxy of the same rank. The proxy C_2 has a rank of 2, because the minimal IP-path is $(C_1 \lhd U_2 \lhd T_2 \rhd C_2)$, which has 2 backward steps. C_2 is not a 2-rank-source, because it is referenced by T_2, which also has a rank of 2.

The rank can be used to determine in which iteration of the *while* loop in Algorithm 1 the proxy can be generated. If the rank of a proxy $v \in V$ is 0 it means that there is an IP-path of v without backward steps. Therefore, the method ExpandBackward is not necessary to generate v and v can be generated before the first *while* iteration. If the rank of v is 1 it means that every IP-path of v has at least one backward step. This implies that v was not generated before the first iteration, because ExpandBackward must be called at least once to generate v.

Suppose RG is a desired reference graph. The idea of the construction of an execution, which generates RG is as follows: A proxy $v \in V$ with $rank(v) = n$ for some $n \in \mathbb{N}_{\geq 1}$ is generated in the n-th iteration of the *while* loop. If the proxy is an n-rank-source, the proxy is generated in ExpandBackward, and otherwise, the proxy is generated by ExpandForward. A proxy $v \in V$ with $rank(v) = 0$ is only created iff $v \notin IP$. The reason is that proxies in *IP* are always retained and directly added to the reference graph, independently of the generation of new proxies. If $rank(v) = 0$ and $v \notin IP$ hold, then the proxy is generated in the first ExpandForward execution before the *while* loop. If a proxy $v \in V$ is a 0-rank-source, then v is not generated by the algorithm at all, because we later prove that $v \in IP$ holds. If a proxy $v \in V$ is not a 0-rank-source, but $rank(v) = 0$ and $v \notin IP$ holds, then v is generated by ExpandForward before the *while* loop. An overview of when each proxy is generated can be seen in Table 4.1. Following these instructions, the reference graph in Figure 4.5 is generated as follows:

1. C_1 is not generated by the algorithm, as it is the initial proxy.
2. No proxies are generated in the first ExpandForward execution (before the *while* loop).
3. U_1 and T_1 are generated in the first iteration of the *while* loop in ExpandBackward.
4. U_2 is generated in the first iteration of the *while* loop in ExpandForward (the second execution of ExpandForward).
5. T_2 is generated in the second iteration of the *while* loop in ExpandBackward.
6. C_2 is generated in the second iteration of the *while* loop in ExpandForward (the third execution of ExpandForward).

We begin the completeness proof by showing some important properties regarding the rank of the proxies. The first property is that proxies cannot reference a proxy of a higher rank. This is fulfilled the example in Figure 4.6, since no reference is pointing upwards.

Lemma 4.7.7 (No Edges to Higher Rank) *Let $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph. Let $e \in E$ be an edge. Then, $rank(t(e)) \leq rank(s(e))$ holds.*

Proof. Let $n = rank(s(e))$ be the rank of $s(e)$. Therefore, there is an IP-Path $p = (v_0, ((v_i, d_i))_{i=1}^{m})$ with $m \in \mathbb{N}$, $v_m = s(e)$, and n backward steps. Let $v_{m+1} = t(e)$ and $d_{m+1} = \triangleright$. Consider the path $p' = (v_0, ((v_i, d_i))_{i=1}^{m+1})$, where the edge e was appended to the path p. Because p is an IP-path for $s(e)$ with n backward steps, p' is an IP-path for $t(e)$ with n backward steps. Therefore, a minimal IP-path for $t(e)$ cannot have more than n backward steps and $rank(t(e)) \leq n$ holds. □

The next property is that every non-empty, minimal IP-path to an n-rank-source ends with backward step. This property is useful to prove the two subsequent lemmas. We use the reference graph in Figure 4.6 as the example again. The minimal IP-path of U_1 is $(C_1 \triangleleft U_1)$, which only has one backward step. The proxy T_2 has two different minimal IP-paths: $(C_1 \triangleleft U_2 \triangleleft T_2)$ and$(C_1 \triangleleft T_1 \triangleright U_2 \triangleleft T_2)$. They are both minimal because we defined minimal with the amount of backward steps and not the total length. Both of these paths end with a backward step.

Lemma 4.7.8 (Backward Step in IP-Path) *Let IP be a set of initial proxies and $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph. Let $v \in V$ be an n-rank-source for some $n \in \mathbb{N}$. Let $p = (v_0, ((v_i, d_i))_{i=1}^{m})$ be a minimal IP-Path of v with $m \in \mathbb{N}_{\geq 1}$. Then, the last step of p is a backward step: $d_m = \triangleleft$.*

Proof. As $m \geq 1$ holds, p is not empty. Assume for the sake of contradiction that the last step $d_m = \triangleright$. Then, there is an edge $e \in E$ with $s(e) = v_{m-1}$ and $t(e) = v_m$. Let $p' = (v_0, ((v_i, d_i))_{i=1}^{m-1})$ be the path p without the last step. As p is an IP-Path, $v_0 \in IP$ holds. Therefore, p' is an IP-Path of v_{m-1}. The path p' has n backward steps and is also minimal, since otherwise, p would not be minimal. This implies that $rank(v_{m-1}) = n$. Therefore, v cannot be n-rank-source, as the edge e exist. This is a contradiction. □

The following property states that every 0-rank-proxy is one of the initial proxies in *IP*. This property is used to justify that we do not generate 0-rank-proxies in the execution we construct. Generating these proxies is not required, as the algorithm adopts them from the set of initial proxies. In the reference graph in Figure 4.6, only C_1 is a 0-rank-proxy. The property is fulfilled, since C_1 is an initial proxy.

Lemma 4.7.9 (0-Rank-Source is in IP) *Let IP be a set of initial proxies, $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph, and $v \in V$ be a 0-rank-source proxy. Then, $v \in IP$ holds.*

Proof. By the definition of an n-rank-source, $rank(v) = 0$ holds. Therefore, an IP-Path $p = (v_0, ((v_i, d_i))_{i=1}^{m})$ with $v_m = v$, $v_0 \in IP$ and without backward steps exists. Assume for the sake of contradiction that p is not empty, so $m \geq 1$ holds. Then, Lemma 4.7.8 implies that d_m is a backward step, which is a contradiction. This implies that $m = 0$. Therefore, we know:

$$\begin{aligned} m = 0 &\implies v_m = v_0 && \\ &\implies v = v_0 && v_m = v \\ &\implies v \in IP && v_0 \in IP \end{aligned}$$

□

We already showed in Lemma 4.7.8 that every non-empty, minimal IP-path to an n-rank-source ends with a backward step. The next-to-last proxy, which is

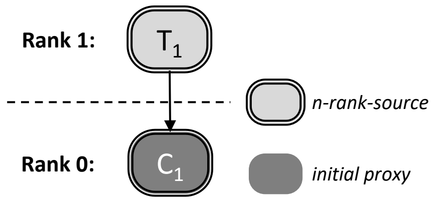

Figure 4.7: The input reference graph RG' before the execution of ExpandForward.

the target of this backward step, always has the rank below $(n-1)$. Therefore, for every n-rank-source with $n \in \mathbb{N}_{\geq 1}$, an edge to a proxy with rank $n-1$ exists. This does not apply to 0-rank-sources, as there is no rank below. In Figure 4.6 the proxies U_1, T_1 and T_2 all have an edge to the rank below.

Lemma 4.7.10 (N-Rank-Source has Edge to Rank Below) *Let IP be a set of initial proxies, $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph and $v \in V$ be an n-rank-source for some $n \in \mathbb{N}_{\geq 1}$. Then, a proxy $w \in V$ with the properties*

1. $rank(w) = n - 1$
2. $\exists e \in E\colon s(e) = v \wedge t(e) = w$

exists.

Proof. By the definition of an n-rank-source, $rank(v) = n$ holds. Thus, a minimal IP-Path $p = (v_0, ((v_i, d_i))_{i=1}^{m})$ for v with n backward steps exist. As $n \geq 1$ holds, p is not empty and $m \geq 1$. We know from Lemma 4.7.8 that d_m is a backward step. Therefore, there is an edge $e \in E$ with $s(e) = v_m$ and $t(e) = v_{m-1}$. Let $p' = (v_0, ((v_i, d_i))_{i=1}^{m-1})$ be the path without the last step. The path p' is an IP-Path of v_{m-1}, as $v_0 \in IP$ holds. As p contains n backward steps and d_m is a backward step, p' contains $n-1$ backward steps. We also know that p' is minimal, since otherwise, p would not be minimal. Therefore, $rank(v_{m-1}) = n - 1$ holds. Due to the edge e, both properties are satisfied for v_{m-1}.

□

Using the previous lemmas, we can now define an important property of ExpandForward. We specify which criteria the input reference graph RG' must fulfil, in order to guarantee that a given reference graph RG can be generated by ExpandForward. As an example, an input reference graph RG' is shown in Figure 4.7 and the corresponding output reference graph RG is shown in Figure 4.8. In order to guarantee that RG can be generated by ExpandForward with RG' as the input, RG' must be a subgraph of RG. In the example, this is the case, as all proxies in RG' are also contained in RG.

We assume that ExpandForward only generates proxies of the same rank in one iteration. This simplifies the proof and does not restrict the reference graphs

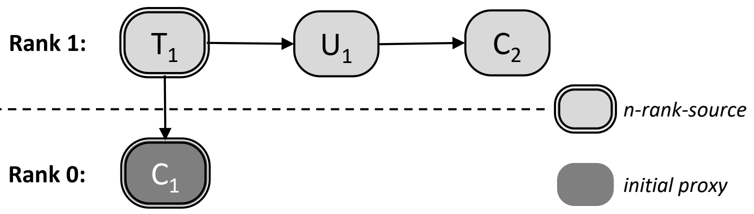

Figure 4.8: The output reference graph *RG* after the execution of ExpandForward.

that can be generated by the algorithm. Assuming that the maximal rank of *RG* is *r*, only proxies with a rank of *r* are added to RG'. This implies that every proxy in *RG* with a rank smaller than *r* already exists in the input graph RG'. In the example, the maximal rank is $r = 1$. Therefore, all proxies in *RG* with a rank of 0 must also be contained in the reference graph RG'. This property is satisfied, since C_1 is the only proxy with a rank of 0 and C_1 is contained in both reference graphs. Additionally, all *r*-rank-proxies of *RG* must be contained in RG'. In the example, T_1 is the only 1-rank-proxy. If T_1 was not contained in the input reference graph, then T_1, U_1 and C_2 could not be generated by the ExpandForward execution.

In the proof, we specify the random choices of ExpandForward that would lead to the generation of *RG*. The random order, in which ExpandForward visits the proxies, must be a topological order. In our example, the only topological order is (T_1, U_1, C_2). Therefore, T_1 is visited first and the proxy U_1 is generated. U_1 is visited afterwards and the proxy C_2 is generated. Finally, the proxy C_2 is visited and ExpandForward terminates.

Theorem 4.7.3 (ExpandForward Property) *Let RS* $= (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$ *be a reference schema, IP be a set of initial proxies and RG* $= (V, E, s, t, \varphi, \psi)$ *be a desired reference graph. Let* $r := \max_{v \in V}(rank_{RG}(v))$ *be the maximal rank in RG. Let* $RG' = (V', E', s', t', \varphi', \psi')$ *be a subgraph of RG with the following properties:*

- $V' \subseteq V$
- $\{ v \in V \mid rank_{RG}(v) < r \} \subseteq V'$
- $\{ v \in V \mid v \text{ is } r\text{-rank-source of } RG \} \subseteq V'$
- $E' \subseteq E$
- $\{ e \in E \mid rank_{RG}(s(e)) < r \} \subseteq E'$
- $(s' = s|_{E'}) \wedge (t' = t|_{E'}) \wedge (\varphi' = \varphi|_{V'}) \wedge (\psi' = \psi|_{E'})$

Let all proxies $v \in V'$ *with* $rank_{RG}(v) = r$ *be marked as unvisited and all proxies* $v \in V$ *with* $rank_{RG}(v) < r$ *be marked as backward_visited. Then, there are random choices such that ExpandForward generates RG when RS and* RG' *are the input.*

Proof. Let V_r be the set of all proxies in RG of rank r:

$$V_r = \{\, v \in V \mid rank_{RG}(v) = r \,\}$$

As RG is acyclic, there exists a topological ordering of RG. Such an ordering of the proxies exists for every subset of V, because still no edge to a previous element in the ordering can exist. Therefore, there exists a list $o \in V_r^{|V_r|}$ with the following properties:

- $\forall i, j \in \{\, 1, \ldots, |V_r| \,\} : i \neq j \Rightarrow o_i \neq o_j$
- $\forall e \in E \colon \forall i, j \in \{\, 1, \ldots, |V_r| \,\} : (s(e) = o_i \wedge t(e) = o_j) \Rightarrow i < j$

This means that it is possible to visit each proxy before any successor of that proxy was visited. Let $n := |V_r|$ be the amount of proxies of rank r. We show that it is possible that ExpandForward visits the proxies in the order of the list $o = (o_1, \ldots, o_n)$, i.e. there are random choices, which lead to this order of visits. Additionally, we show that RG can be generated, when the proxies are visited in this order. To achieve this, we use a loop invariant proof:

Loop Invariant: Before the k-th iteration and after the $(k-1)$-th iteration of the *while* loop, it is possible that all the following statements hold:

1. Precisely the following proxies were visited:
$$\{\, o_1, \ldots, o_{k-1} \,\}, \tag{4.3}$$
2. Precisely the following proxies were generated:
$$\{\, v \in V \mid \exists e \in E \colon s(e) \in \{\, o_1, \ldots, o_{k-1} \,\} \wedge t(e) = v \,\} \setminus V' \tag{4.4}$$
3. Precisely the following references were generated:
$$\{\, e \in E \mid s(e) \in \{\, o_1, \ldots, o_{k-1} \,\} \,\} \setminus E'. \tag{4.5}$$

Initialization: Let $k = 1$. Before the first iteration of the *while* loop in Expand-Forward no proxies were visited, no proxies were generated, and no edges were generated. This corresponds to the statements in the loop invariant:

1. Visited proxies: $\{\, o_1, \ldots, o_0 \,\} = \emptyset$
2. Generated proxies: $\{\, v \in V \mid \exists e \in E \colon s(e) \in \emptyset \wedge t(e) = v \,\} \setminus V' = \emptyset$
3. Generated references: $\{\, e \in E \mid s(e) \in \emptyset \,\} \setminus E' = \emptyset$

Maintenance: We assume that the three statements of the loop invariant are satisfied before the k-th iteration. In order to show that it is possible that these statements still hold after the k-th iteration, we specify random choices, which lead to the fulfilment of these statements.

The first random decision is, which of the unvisited proxies to visit next. First, we show that it is possible that the algorithm chooses the proxy o_k in the k-th iteration. This requires that o_k is currently in the set of unvisited proxies. There are two different cases:

Case 1: $o_k \in V'$. By the assumption of the theorem, every proxy $v \in V'$ with $rank_{RG}(v) = r$ is marked as *unvisited*. Therefore, o_k was unvisited at the beginning of the ExpandForward execution. Because $o_k \notin \{ o_1, \dots, o_{k-1} \}$, we know from Equation 4.3 it has not been visited before in this Expand-Forward execution. This implies that o_k is in the set of unvisited proxies.

Case 2: $o_k \notin V'$. Because $o_k \in \{ o_1, \dots, o_n \}$, we know that $rank_{RG}(o_k) = r$. Since $o_k \notin V'$, the definition of V' implies that o_k is not an r-rank-source of RG. The fact $rank_{RG}(o_k) = r$ and the definition of r-rank-source imply that a reference $e \in E$ exists with $rank(s(e)) = r$ and $t(e) = o_k$. Since $rank(s(e)) = r$, we know that $s(e) \in \{ o_1, \dots, o_n \}$. As o is topologically ordered and $s(e)$ references o_k, $s(e) \in \{ o_1, \dots, o_{k-1} \}$ holds. The second property of the loop invariant in Equation 4.4 and the fact $o_k \notin V'$ imply that o_k was generated. Because $o_k \notin \{ o_1, \dots, o_{k-1} \}$, it has not been visited before in this ExpandForward execution. This implies that o_k is in the set of unvisited proxies.

Therefore, o_k is in the set of unvisited proxies and can be visited in the k-th iteration.

We assume that the proxy o_k is chosen in the k-th iteration and show that it is possible that the statements of the invariant hold after the k-th iteration. The next random choice is the Boolean value *doesGenerateOptional*. We assume *does-GenerateOptional* to be true if and only if *dependency* is satisfied for the proxy *selected* in RG. This is possible, since the value is chosen randomly. Therefore, an optional reference is only created iff it exists in E. If the dependency is required, the *if* block is always executed, as this is checked in the *if* condition. This makes sense, as RG is desired and therefore fully defined. This implies that there is an edge in E, which satisfies *dependency* for the proxy *selected*. So whenever the *if* block is executed, there is an edge $e \in E$ with $dependency = \varphi(e)$ and $s(e) = selected$.

We assume that the next Boolean value *useExistingProxy* is true iff $t(e) \in POT$. This is possible, since this value is also chosen randomly. If $t(e)$ does not exist already, then it cannot be in POT and $t(e)$ is newly created. The case that $t(e)$ already exists, but is marked as *forward_visited*, is impossible, as that would mean, that $t(e)$ was visited before $s(e)$. This can never happen, as we visit the proxies in the order of the list o. If $t(e)$ already exists and is marked as *unvisited* or *backward_visited*, then an element from *POT* is chosen as the target of the new edge. In this case, we assume that the algorithm picks $t(e)$ as the random target. This is possible, since $t(e)$ is marked as *unvisited* or *backward_-visited*, and therefore $t(e) \in POT$ holds. Thus, the algorithm only creates edges that are in E and proxies which are in V. So no additional unnecessary edges or proxies are created.

The set of proxies generated in the previous iterations of the *while* loop is known from the loop invariant:

$$V_{prev} = \{ v \in V \mid \exists e \in E\colon s(e) \in \{ o_1, \dots, o_{k-1} \} \wedge t(e) = v \} \setminus V'$$

Provided the mentioned random values were chosen, the set of proxies generated in this iteration is:

$$V_{new} = \{ v \in V \mid \exists e \in E\colon s(e) = o_k \wedge t(e) = v \} \setminus (V_{prev} \cup V').$$

The set of all generated proxies by ExpandForward is:

$$V_{new} \cup V_{prev} = \{ v \in V \mid \exists e \in E\colon s(e) \in \{ o_1, \dots, o_n \} \wedge t(e) = v \} \setminus V'.$$

Therefore, the second property of the loop invariant is fulfilled. For brevity, we omit the definition of the set of generated references. The new set of visited proxies is $\{ o_1, \dots, o_k \}$, since only o_k was visited in this iteration.

Termination: We assume that the three statements of the loop invariant are satisfied after the n-th iteration. This is possible, due to the loop invariant. To prove that the algorithm terminates after n iterations, we have to show that no proxies are unvisited. Let $v \in V$ be a proxy.

Case 1: $rank_{RG}(v) < r$. Due to the definition of V', $v \in V'$ holds. By the theorem assumption, all proxies in V' with $rank_{RG}(v) < r$ are marked as *backward_-visited*.

Case 2: $rank_{RG}(v) = r$. By the definition of V_r, $v \in V_r$ holds. Therefore, v is in $\{ o_1, \dots, o_n \}$. Equation 4.3 from the loop invariant states that v was visited.

Therefore, all proxies in V are visited. Additionally, we have to show that all proxies $V \setminus V'$ and all references $E \setminus E'$ were generated. We know that all proxies in $V \setminus V'$ were generated, since we showed that all proxies in V are visited. Let $e \in E \setminus E'$ be a reference.

$$\begin{aligned}
e \in E \setminus E' &\implies rank_{RG}(s(e)) = r && \text{Def. } E' \\
&\implies s(e) \in V_r && \text{Def. } V_r \\
&\implies s(e) \in \{ o_1, \dots, o_n \} && \text{Def. } o \\
&\implies e \text{ was generated} && e \notin E' \text{and Equation 4.5}
\end{aligned}$$

□

The idea of the completeness proof is to show that all proxies with a rank of n can be generated in the first n iterations of the *while* loop of Algorithm 1. An edge is generated as soon as both the source and the target of the edge were generated. To formally define the intermediated reference graphs, we use a concept called induced subgraph:

Definition 4.7.4 (Induced Subgraph). Let $RG = (V, E, s, t, \varphi, \psi)$ be a reference graph and $V' \subseteq V$ be a subset of the proxies. Then, the induced subgraph $RG[V'] = (V', E', s', t', \varphi', \psi')$ is a reference graph with the following properties:

1. $E' = \{ e \in E \mid s(e) \in V' \wedge t(e) \in V' \}$
2. $(s' = s|_{E'}) \wedge (t' = t|_{E'}) \wedge (\varphi' = \varphi|_{V'}) \wedge (\psi' = \psi|_{E'})$

As the set V', which defines the content of the induced subgraph, we use the set of proxies with a specific rank and below. So for a given rank r, we use $V' = \{ v \in V \mid rank(v) \leq n \}$ as the set of proxies. For the reference graph in Figure 4.6 this means that for a rank $r = 1$ the set of proxies is $\{ C_1, U_1, T_1, U_2 \}$. The following property states that the induced subgraph which only contains these proxies is desired.

Lemma 4.7.11 (Induced Subgraph is Desired) *Let $RS = (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$ be a reference schema, IP be a set of initial proxies, $\varphi_0 : IP \to \mathcal{V}$ be a function, which assigns types to the proxies in IP and $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph. Let $n \in \mathbb{N}$ and $V' = \{ v \in V \mid rank(v) \leq n \}$. Let $RG' = RG[V']$ be an induced subgraph of RG. Then, RG' is desired.*

Proof. This lemma states that a desired reference graph stays desired when all proxies with a specific minimal rank or above are removed. For brevity, we only prove the most important property of *desired*, which is that RG' is still fully defined.

Let $v \in V'$ be a proxy. Let $d \in \mathcal{E}$ be a required dependency of v. As RG is fully defined, an edge $e \in E$ with $s(e) = v$ and $\psi(e) = d$ exists. Assume for the sake of contradiction that $e \notin E'$. Then, $t(e) \notin V'$ and $rank_{RG}(t(e)) > n$ holds by the definition RG'. Since $e \notin E'$ holds, we know that $rank(s(e)) \leq n$. This is a contradiction, due to Lemma 4.7.7. Therefore, $e \in E'$ holds, and RG' is fully defined. □

We prove the completeness property by induction on the iteration of the *while* loop. In the n-th iteration, all desired reference graphs with a maximal rank of n can be generated.

Theorem 4.7.4 (Completeness) *Let $RS = (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$ be a reference schema, IP be a set of initial proxies, $m \in \mathbb{N}$ be an amount of iterations, and $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph with $\max_{v \in V} \left(rank_{RG}(v) \right) \leq m$. Then, RG can be generated by Algorithm 1 with RS, IP and m as the input.*

Proof. Let $RS = (\mathcal{V}, \mathcal{E}, sourceType, targetType, necessity)$ be a reference schema, IP be a set of initial proxies and $m \in \mathbb{N}$. We will only show that every reference graph with a maximal rank of n with $n \leq m$ can be generated in n iterations of the algorithm. This also implies that every reference graph can also be generated in m iterations, even if $m > n$. The reason is that it is possible that all subsequent iterations do not alter the reference graph. This would happen if ExpandBackward always chooses 0 as the amount of proxies to add. The method ExpandForward would also not add any proxies, as there are no *unvisited* proxies to visit. We prove the statement by induction on the maximal rank of the reference graph:

Base Case: Let $n = 0$. Let $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph with

$$\max_{v \in V} \left(rank_{RG}(v)\right) = n.$$

Therefore, all proxies have a rank of 0. Let RG' be the initial reference graph with only the initial proxies IP. We know from Lemma 4.7.9 that every 0-rank-source proxy of RG is in IP. This means that all 0-rank-proxies of RG are in V'. We now know from Theorem 4.7.3 that RG can be generated by the first ExpandForward execution. No proxy is marked as *unvisited*, as this is a postcondition of ExpandForward. There is also no proxy marked as *backward_visited*, as ExpandBackward was never called. Therefore, all proxies are marked as *forward_visited*.

Induction Hypothesis: Given some $n \in \mathbb{N}$ with $n \leq m$, then every desired reference graph with

$$\max_{v \in V} \left(rank_{RG}(v)\right) = n$$

can be generated in n iterations of the *while* loop of the algorithm such that every proxy of rank n is marked as *forward_visited* and all other proxies are marked as *backward_visited*.

Induction Step: Let $RG = (V, E, s, t, \varphi, \psi)$ be a desired reference graph with

$$\max_{v \in V} \left(rank_{RG}(v)\right) = n + 1.$$

Let V_{prev} be defined as

$$V_{prev} = \{\, v \in V \mid rank_{RG}(v) < n + 1 \,\}.$$

Let RG_{prev} be the induced subgraph $RG\left[V_{prev}\right]$. We know from Lemma 4.7.11 that RG_{prev} is desired. V contains proxies of rank n, because there exist IP-Paths to proxies of rank $n + 1$. Therefore,

$$\max_{v \in V_{prev}} \left(rank_{RG}(v)\right) = n$$

holds. Now we know from the Induction Hypothesis that RG_{prev} can be generated in n iterations. We show that in the next iteration of the *while* loop, RG can be generated.

First, we specify which random choices in the ExpandBackward method lead to the generation of all $(n+1)$-rank-source proxies. The only random choice of this method is how many backward references are created per proxy. We know from Lemma 4.7.10 that for each $(n+1)$-rank-source proxy $v \in V$, there is a successor $w \in V$ of v with $rank_{RG}(w) = n$. Since $rank_{RG}(w) = n$ holds, we know that $w \in V_{prev}$ holds, and w is contained in RG_{prev}. By the induction hypothesis, w is marked as *forward_visited* and therefore visited in ExpandBackward. When w is visited, the backward reference to v can be generated, as an arbitrary amount of proxies is generated for every dependency of w. Therefore, all $(n+1)$-rank-source proxies can be generated. The proxy v might have multiple successors w, but the proxy v shall only be generated once. However, since the amounts of generated proxies are random, it is possible that the backward reference to v is only created for one of the successors. In this case, the set of newly generated proxies is:

$$V_{new} = \{\, v \in V \mid v \text{ is } (n+1)\text{-rank-source of } RG \,\}$$

Let $RG' = (V', E', s', t', \varphi', \psi')$ be the returned reference graph from ExpandBackward assuming the specified random choices are made. Our goal is to use Theorem 4.7.3 to show, that RG can be generated in the next ExpandForward execution with RG' as the input. Therefore, we have to show that the assumptions of Theorem 4.7.3 are fulfilled. We know that $V_{prev} \subseteq V$ and $V_{new} \subseteq V$ hold. $V' = V_{prev} \cup V_{new}$ implies:

$$V' \subseteq V$$

The definition of V_{prev} and the fact $V_{prev} \subseteq V'$ imply:

$$\{\, v \in V \mid rank_{RG}(v) < n+1 \,\} \subseteq V'$$

The definition of V_{new} and the fact $V_{new} \subseteq V'$ imply:

$$\{\, v \in V \mid v \text{ is } (n+1)\text{-rank-source of } RG \,\} \subseteq V'$$

Since RG_{prev} is an induced subgraph of RG, all edges in E_{prev} are also contained in E. We assumed that ExpandBackward only generates edges in E. Therefore, we know:

$$E' \subseteq E$$

The definition of V_{prev}, the definition of induced subgraph, and Lemma 4.7.7 imply:

$$\begin{aligned}
&\{\, v \in V \mid rank(v) \leq n \,\} = V_{prev} && \text{Def. } V_{prev}\\
\Longrightarrow\ &\{\, e \in E \mid rank(s(e)) \leq n \wedge rank(t(e)) \leq n \,\} = E_{prev} && \text{Def. ind. subgr.}\\
\Longrightarrow\ &\{\, e \in E \mid rank(s(e)) \leq n \,\} = E_{prev} && \text{Lemma 4.7.7}\\
\Longrightarrow\ &\{\, e \in E \mid rank(s(e)) \leq n \,\} \subseteq E' && E_{prev} \subseteq E'
\end{aligned}$$

Therefore, we know from Theorem 4.7.3 that RG can be generated in the following ExpandForward execution. All proxies v with $rank_{RG}(v) \leq n$ are marked as *backward_visited*, as they were either already *backward_visited* in the previous iteration or were visited in ExpandBackward in the current iteration. The proxies with rank $n + 1$ have been created in the current iteration, and therefore cannot have been visited by ExpandBackward. As no proxies can be marked as *unvisited* when ExpandForward terminates, these proxies are marked as *forward_visited*.

□

4.8 Discussion

In this section, the advantages, disadvantages and limitations of the algorithm are discussed.

4.8.1 Termination

An important property of the algorithm, which was not discussed previously, is its termination. The algorithm is not guaranteed to terminate under all conditions. Consider the case that the reference schema contains a directed cycle of which all edges are required. We call a cycle of this kind a required cycle. It corresponds to a referential cycle of database tables, where each of the foreign keys is not nullable. Every reference graph, which contains at least one proxy in the required cycle of the reference schema, therefore contains a cycle. However, the algorithm we defined can only return acyclic reference graphs, so it can never return. We do not consider this a notable limitation, as required cycles are very problematic in general. For Azure SQL databases these cycles are even prohibited [Azure, 2010]. A possible refinement of the algorithm is to check if the reference schema has required cycles in advance, and possibly return a corresponding error message.

A more interesting case is a referential cycle in the database, where at least one of the foreign keys is nullable. We call such a cycle an optional cycle. If a reference schema contains an optional cycle, the algorithm as it is currently defined also does not guarantee termination, as the algorithm might randomly decide to always create a reference, although it is optional. This problem can be fixed with a small change: When the reference graph already reached a certain size, references are no longer created when they are optional. Therefore, at some point only required references are created and the algorithm terminates.

A reasonable change to the algorithm would be to have a different termination condition for the *while* loop. Currently, the *while* loop terminates after a predefined amount of iterations. However, in some cases, it would make sense to define a maximal amount of proxies instead. But if the size threshold is a maximal amount of proxies in the reference graph, it is possible that in one iteration of the *while* loop no further proxies are generated. In this case, the algorithm makes no progress towards the size threshold. Therefore, termination cannot be guaranteed anymore. One way to solve this problem is to terminate if the algorithm did not increase the size of the reference graph in one iteration. In general, the termination condition must be chosen carefully.

4.8.2 Restrictions

The algorithm we proposed only generates *desired* reference graphs. The definition of *desired* already implies two limitations: The first limitation is that only reference graphs are generated, where every proxy is connected to one of the initial proxies. This is an intentional decision to avoid the generation of database rows, which are completely unrelated to the database rows, we wanted to generate. Consider the reference schema in Figure 4.1, and suppose we want to generate a company to test its behaviour. To achieve this, the set of initial proxies can be defined as a single proxy of type *Company*. In most cases, the existence of a user only makes a difference to the behaviour, if it is in some way connected to the company. For example, the connection could be that the user works at the company or is assigned to a task of this company. Thus, the algorithm does not generate unrelated users. However, there are exceptions to that rule. For example, if the tested function searches for potential employees for the company, then the existence of unrelated users can influence the result. It is possible to solve this problem, by adding a random number of proxies of type *User* to the initial proxy set beforehand. This example shows that it is important to carefully define the initial proxy set, as otherwise corner cases may be missed.

The second limitation caused by only generating *desired* reference graphs is that only acyclic reference graphs are generated. This is also an intentional decision, as it is often considered a good practice to avoid relationship cycles of database entries. However, this does not mean that we prohibit cycles on a table level. We will introduce an example, where allowing cycles on a row-level can lead to problems: Let's consider an approach to store a list-like structure in a relational database. There is a database table *Element* with an optional field *successorId*, which points to the next element in the list. If cycles were allowed, an element could point to itself. Semantically, this would mean that the list is infinitely large and has no last element. You would run into similar issues when defining tree-like structures. Another problem is the insertion of cyclic content into the database. When the content is acyclic, each row can be inserted before it is referenced, so no integrity constraint violations occur. This is not possible when the content contains cycles. Even when all rows are inserted in the same transaction, integrity constraint violations can occur, as in some database management systems like *PostgreSQL* and *Oracle*, the foreign key constraints are checked after each insert by default. This can be changed by adding *DEFERRABLE INITIALLY DEFERRED* to all relevant foreign keys, which means that these constraints are only checked after the transaction is completed. However, it would be a significant limitation, if the test case generation only worked on databases, where the foreign keys are labelled as *DEFERRABLE INITIALLY DEFERRED*. For these reasons, we decided to generate acyclic reference graphs only. Allowing self-references but no larger cycles would still avoid the problem of inserting the content, but we still decided against it.

4.8.3 Constraints

We will now discuss, which constraints can be expressed using our approach and which cannot. We divide constraints into three categories: The first category is *foreign key constraints*. They can be expressed with our approach, as the *foreign key constraints* are integrated into the reference schema and are used to generate references. We call the second category *local constraints*, which are constraints that only depend on the content of a single row. Examples are that a numeric value has to be in a specified range or a *NOT NULL* constraint. *Local constraints* are not modelled into the reference schema. But our approach only generates proxies, which later should be replaced with actual database rows with content. The generation of these rows should take *local constraints* into account in order to satisfy them. We call constraints which do not fit in the first two categories *complex constraints*. In general, *complex constraints* cannot be expressed using our approach, which is one of the main limitations of our approach. An example is that every company must have at least five employees. As these constraints can get arbitrarily complex, a constraint solver is required to satisfy them reliably. There are approaches of test case generation using constraint solvers, however, we considered this to be beyond the scope of this thesis.

4.9 Implementation

We implemented the proposed algorithm in Scala in a project-independent library. As a proof of concept, the library was used to generate test data for a real-world application. The reference graph generator uses the writer approach from Chapter 3, so the output of the algorithm is a `GenWithDBActions`. Therefore, the generation and insertion of the values are both encapsulated into a single monad. This has the following benefits:

- The generation of the graph is completely seed-based. If a test fails, the failure can be reproduced by generating the same graph with the same seed again.

- The reference graph generator can be combined with other generators. Therefore, a previous generator can randomly generate the parameters of the reference graph generator. Additionally, subsequent generators can generate the input depending on the values which are inserted into the database.
- The generated database content can easily be inserted into the database in a single transaction. For example, the `forAllDB` method, which is described in Section 3.3, can be used, in order to insert all generated content automatically before the property is evaluated.

4.9.1 HList

The generator returns the list of generated database rows. These are useful to define the correct behaviour of a method. However, more important than all rows, which were generated, are the generated rows, which correspond to the initial proxies. For example, consider a set of initial proxies, which contains one `Proxy[User]` and one `Proxy[Company]`. The algorithm potentially creates a lot of database rows, but the most interesting part is the `User`, which corresponds to the `Proxy[User]` and the `Company`, which corresponds to the `Proxy[Company]`. One possible approach is to separately return the set of all database rows which correspond to one of the initial proxies. If the tester is interested in one specific row, they could iterate over the set to find it. The type of the set would have to be `Set[Any]`, as the elements can have arbitrary types. This approach is flawed, as the elements cannot be accessed in a type-safe manner and the tester has to search for the values they are interested in. To solve this problem, we utilized a concept called `HList` (heterogeneous list). An `HList` is a list of values with different types, where the type of each element of the list is already known at compile-time. At the same time, they are more powerful than tuples, as it is possible to define operations as `.map()` on them, without knowing the size. In our implementation, the initial proxies are modelled as an HList. The type of the HList could be `Proxy[User] :: Proxy[Company] :: HNil`, which means the list contains two elements: The first element is of type `Proxy[User]`, and the second element is of type `Proxy[Company]`. In this case, the initial proxy list can be defined as follows:

```
val userProxy: Proxy[User] = ???
val companyProxy: Proxy[Company] = ???

val initialProxies = userProxy :: companyProxy :: HNil
```

As the output of the algorithm has the type `GenWithDBActions`, a property can be defined using the `forAllDB` method from the writer approach.

```
val graphGen = ReferenceGraphGenerator.gen(initialProxies)
val prop = forAllDB(graphGen) {
    case ProxySetup(initialRows, allRows) =>
        val user: User = initialRows(_1)
        val company: Company = initialRows(_2)

        //test something
}
```

The generated database rows which correspond to the initial proxies are represented as an `HList` called `initialRows`. As the types of the initial proxies are known at compile-time, the types of the generated values in `initialRows` can be derived to be `User :: Company :: HNil` at compile-time. This makes it possible to access these generated values in a type-safe manner.

4.9.2 Templates

The reference graph generator takes the values that should be generated as input. Therefore, the tester can specify requirements like *"two users U_1 and U_2 and a company C_1 shall be generated"*. However, this does not specify in which relationship the generated values are. The generated users and the company can either be connected or be independent of each other. In some test scenarios, the relationship between the generated values is essential. To address this issue, a *template* can be defined for proxies. The template contains some predefined references of the proxy, which are taken into account by the algorithm. This makes it possible to specify the requirement *"two users U_1 and U_2 that work at the same company C_1 shall be generated"*. This can be viewed as having an *initial references* set *IR* additionally to the *initial proxies* set *IP*.

4.9.3 Configuration

As a preliminary step, the tester has to define the following methods, which determine how proxies of a given type are generated:

genRefs Defines how forward references are generated for a proxy type. Here, the tester can specify the probability that optional references are generated. For example, it can be specified that only for 20% of the generated users a company is referenced. Additionally, the tester can control the probability of the choice, whether existing or new proxies are referenced.

genBackRefs Defines how many backward references are generated for a given type. For example, the tester can specify that for each company between 10 and 20 employees are generated. This customization is important, as generating many backward references for all proxies can result in larger reference graphs than necessary. It requires domain knowledge to decide, which backward references are relevant, and which are not.

gen Specifies how content for the attributes of the current database row is generated. For example, the tester can specify that the age of users is always above 18. The tester should take database invariants and constraints, which only affect the current row, into account, when they define this method.

insertable Defines how the generated values can be inserted into the database. The `Insertable` type, which is described in Section 3.3, can be used to simplify the definition.

Chapter 5

Stateful Property-Based Testing

Testing the behaviour of a stateful system has two main challenges[Chays et al., 2000]:

Controllability Putting a system into the desired state before execution of a test case

Observability Observing its state after execution of the test cases

The controllability can be dealt with by generating data and inserting it into the database before a test is executed. The writer approach in Chapter 3 and the reference graph generator in Chapter 4 are approaches which solely focus on this challenge. The challenge of observability is more complex than it might seem. The goal is to verify that the tested functionality made the correct changes to the database. While it is easy to check whether all expected changes were made, it is harder to verify that no additional undesirable changes were made. Comparing the complete database state before and after the execution is unreasonably complex.

The approach of stateful property-based testing, which was described in Section 2.3, addresses the challenges of controllability and observability differently: It treats the system as a black box and ignores the fact that a database is involved. The system is brought into the state by the generation and the execution of a random sequence of commands. If a single command is viewed as a test case, all previous commands are responsible for the controllability of that test case. The observability is solved implicitly: By testing whether all subsequent commands behave as expected, it is indirectly checked, whether the system was brought into the correct state. Because when the current command brings the database into an undesired state, subsequent commands likely return different values than expected. Therefore, each command can simultaneously address both challenges.

In the context of this thesis, we used this approach to test two different projects: The first one is a protocol system called *Cap3 Protokollsystem*[1], and we

[1]`https://protokollsystem.de/`

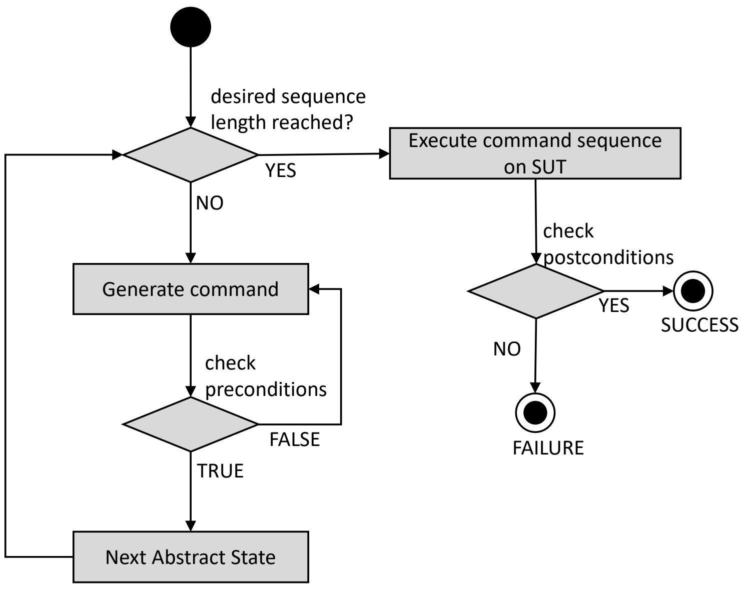

Figure 5.1: A state diagram, which visualizes the generation of test cases using the *offline testing* technique.

modelled a part of the application with 14 commands. The second one is an internal time tracking project called *Captre*, in which we tested most of the project with 11 commands. There are already diverse case studies of stateful property-based testing using either the QuickCheck or ScalaCheck framework, which already show that the approach is effective at finding bugs (see Section 6.2). For this reason, we do not present the projects and our tests in detail, but solely discuss our experiences and the challenges we encountered.

5.1 Problem of Non-Determinism

Many real-world applications contain non-deterministic or unpredictable behaviours. Examples are the usage of timestamps, the generation of UUIDs and the generation of tokens. The generation of UUIDs and tokens are randomized by design and the value of a timestamp can be unpredictable from an outside perspective, since the scheduler cannot be controlled. The goal of the abstract model is to define the intended behaviour of the application, but in these cases, there are many different possible behaviours and not a single correct one. Depending on which of two techniques is used for the test case generation, this can present challenges. With the first technique – called *offline testing* [Veanes et al., 2008] – the test generation and the test execution are independent phases. In Figure 5.1 we show an exemplary state diagram for this technique. The command sequence is generated using exclusively the abstract model before any command was executed on the system under test. We demonstrate how this can present challenges in an example application that consists of two commands:

```
def createUser(name: String): UUID // non-deterministic

def getUserById(id: UUID): User
```

The command `createUser` can be executed without prerequisites and returns the ID of the generated user. However, the command `getUserById` always fails, unless it is called with the ID of a previously generated user. As the abstract model cannot predict, which ID will be created by the system under test, no valid command sequence can be created, beforehand. A small change to the commands solves the problem:

```
def createUser(id: UUID, name: String): Unit

def getUserById(id: UUID): User
```

By passing the ID to the `createUser` method, the internal behaviour becomes deterministic and can be predicted. Now, the abstract model can specify which ID should be used, and therefore knows which is required by the `getUserById` method.

The second technique used for the test case generation is called *online testing* or *on-the-fly testing*[Veanes et al., 2008] and is visualized in Figure 5.2. In this technique, the test generation and the test execution happen in the same phase. Instead of generating the entire command sequence at once, one command is generated and executed at a time. The result of the executed command influences the abstract model, and therefore also the generation of the subsequent commands. When a part of the application is non-deterministic, the abstract model can observe the non-deterministic choices in the results of the execution. Therefore, the non-deterministic behaviour no longer presents the demonstrated challenges.

While QuickCheck uses the online-based technique, ScalaCheck – the framework we use – employs the offline-based approach. One workaround for this problem is to move all non-deterministic behaviours out of the core of the application logic, as in the example above. Then, the core of the application is purely deterministic and can be tested using this approach. The part of the application, which makes the non-deterministic choices, is not tested. In many cases, this is not a problem, as this part contains no application logic. However, the drawback of this workaround is that the implementation of the system under test has to be altered for the test to work, which may be undesirable. Scenarios in which the non-deterministic behaviour of the application cannot be clearly separated from the application logic are also problematic. In the projects we tested, this workaround was already sufficient, as most of the non-deterministic behaviour was already separated from the application logic.

Andersson and Lindbom [2017] also ran into the problem of non-determinism when testing with ScalaCheck. One workaround they propose is to let the abstract model create its own independent abstract identifier for each command

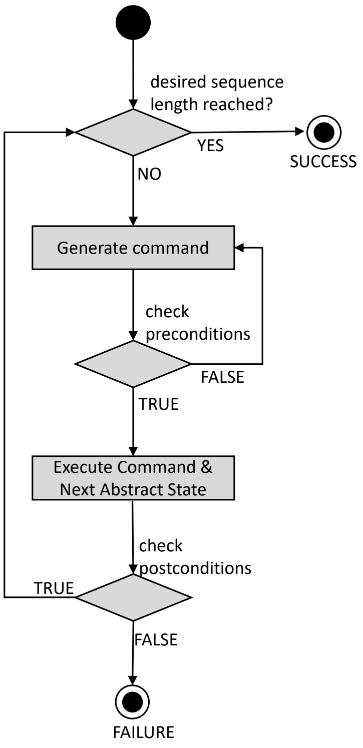

Figure 5.2: A state diagram, which visualizes the generation of test cases using the *online testing* technique. Based on a diagram by Arts and Castro [2011].

that returns an identifier. When the command is executed, a mapping between the abstract identifier and the actual identifier, which is returned by the system under test, is created. This mapping is later used to compare the behaviour of the abstract model with the behaviour of the system under test. This workaround solves the issue of non-deterministically chosen identifiers. However, other non-deterministic behaviours still present challenges. Therefore, they tested a part of the application using the *online testing* approach. There already exists an issue in the ScalaCheck repository regarding this problem, but no solution is implemented at the time of writing[2].

5.2 Challenge of Command Statistics

When an application is tested using stateful property-based testing, it may be useful to see how often each command is executed. In particular, it is important to know for the tester, whether commands, which have strong preconditions and can only be executed in rare scenarios, are sufficiently tested. Additionally, if a mistake in the implementation of the precondition is made, a particular command may never be executed. It can also occur that a command is never executed due to a bug in the precondition. Most property-based testing frame-

[2]`https://github.com/typelevel/scalacheck/issues/199`

works have a feature to classify test cases in order to collect statistics about the generated data. As an example, we assume the following command sequences were generated:

```
sequence 1: [Deposit, Withdraw, Withdraw, GetBalance, Withdraw]
sequence 2: [Deposit, Withdraw, Withdraw]
sequence 3: [Deposit, GetBalance]
```

Ideally, the statistics look something like this:

```
scala> prop.check
+ OK, passed 100 tests.
> Collected test data:
50% Withdraw
30% Deposit
20% GetBalance
```

To use the ScalaCheck statistics feature, the tester can use the method `classify` to classify a test case with a label. One obvious approach is to classify a test case with the name of the command each time a command is executed. ScalaCheck then internally collects all labels that occurred in one test execution in a label set. In the context of stateful property-based tests, one test execution is the execution of an entire command sequence. Therefore, the collected label sets are as follows:

```
label set 1: {Deposit, Withdraw, GetBalance}
label set 2: {Deposit, Withdraw}
label set 3: {Deposit, GetBalance}
```

Here, some information is already lost, as the labels are stored in a set. The set only represents which labels occurred, and not in which order and how often each label occurred.

After all test executions, ScalaCheck prints, for each label set, how often this label set occurred. This leads to the following output:

```
scala> prop.check
+ OK, passed 100 tests.
> Collected test data:
33% Deposit, Withdraw, GetBalance
33% Deposit, Withdraw
33% Deposit, GetBalance
```

This is contrary to what we want, as ScalaCheck does not show, how often a single command was executed, but how often each combination of commands was executed. In a more realistic setting with numerous commands, long command sequences, and numerous test executions, ScalaCheck prints many incomprehensible combinations of commands, where each combination of commands only occurs a single time.

It is not easy to change the classify-approach for stateful property-based testing without changing the classify-approach in general. The entire stateful property-based test is implemented as a regular property and is therefore executed just like other properties. One approach we considered is generating a UUID each time a test case is classified with a command. The command in combination with the UUID is then used as the label. Due to the uniqueness of UUIDs, duplicate commands are no longer removed from the label set. We implemented an alternative approach, where the entire sequence of generated commands is put into a single label. After the execution of all command sequences, these labels are cumulated and evaluated. This way we were able to gain the information, how often each command was executed. However, due to the experimental nature of our implementation, we refrained from pursuing it further.

5.3 Bug in Shrinking Algorithm

When ScalaCheck finds a failing command sequence, it tries to minimize the counterexample in a process called shrinking. We noticed that there are scenarios, where shrinking leads to invalid command sequences, in which not all preconditions of the commands are satisfied. However, it turned out to be a known bug. At the time of writing, a fix for the bug is in progress[3]. We switched to an earlier version of ScalaCheck, where the bug does not exist.

5.4 General Experiences

Using the stateful property-based testing approach, we were able to find a bug in a real-world application virtually without additional effort a week before a release. Additionally, more minor issues and unexpected behaviours were found. We noticed that the issues found by the property-based approach only occur in scenarios, which likely would not have been explicitly tested in a conventional unit test. This demonstrates the advantage of this approach: The test is not limited by the thoroughness and creativity of the tester, due to the randomized nature. Setting up the abstract model and the testing infrastructure is an initial effort overhead, but adding more commands was in most cases straightforward. We also noticed that the shrinking of the command sequence can be surprisingly useful. The shrunken command sequence is often minimal and can be viewed as a list of steps to reproduce the bug, which helps a lot to isolate the source of the problem.

[3]`https://github.com/typelevel/scalacheck/pull/739`

5.5 Hybrid Approach

In this section, we discuss the idea of generating an initial state of a stateful property-based test as a preliminary step. For this purpose, the writer approach in Chapter 3 and the reference graph generator in Chapter 4 can be used. As we already discussed in the beginning of this chapter, the stateful property-based testing approach addresses the challenge of controllability implicitly by executing a random sequence of commands. This makes it seem unnecessary to generate an initial state of the system additionally. However, there are still reasons to put the system into a specific state as a preliminary step:

- Databases may contain legacy data, which cannot be generated with the current API anymore. When the commands represent the current state of the API, it is not tested if the system still works for the legacy data. Therefore, it makes sense to intentionally fill the database with legacy data and test whether the system still behaves as expected afterwards.
- There are stateful property-based tests that only cover one part of an application. In those cases, it can be sensible or even necessary to insert data, which cannot be created by that part of the application, into the database beforehand. This makes a lot of sense in large applications, where a test that covers the entire application is neither desired nor feasible.
- In test cases, where large amounts of data should be generated, it is far more efficient to generate and insert all data at once, instead of generating a long sequence of commands and executing all commands. This is particularly important when there are numerous commands, and each command only has a slight impact on the current state.
- Some commands have very specific preconditions, which are rarely satisfied. Therefore, it cannot be guaranteed if and how often a test case with that command is executed. This problem can be solved by manually bringing the system into a specific state and generating random command sequences from there on.

In stateful property-based tests, the abstract model always has to match the real system. If the database is initially filled with data, the abstract model frequently has to contain a representation of that data as well. There are two ways to achieve this with randomly generated data: The first approach is to use database queries after the insertion to collect the data which should be contained in the abstract model. This approach works independently of the data generation but requires the additional laborious step of querying the data from the database. The cleaner approach is to collect the data while it is generated. In general, this is unproblematic when the data generation is separated from the

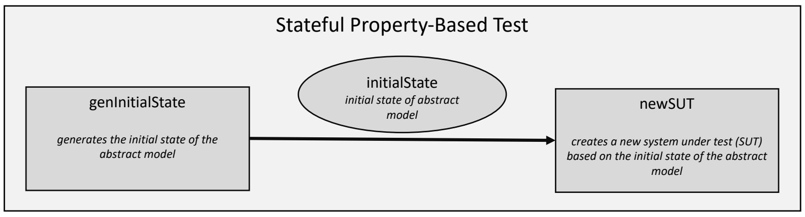
Stateful Property-Based Test
genInitialState
generates the initial state of the abstract model
initialState
initial state of abstract model
newSUT
creates a new system under test (SUT) based on the initial state of the abstract model

Figure 5.3: The `genInitialState` method generates the initial state of the abstract model. ScalaCheck passes the initial state to the `newSUT` method at a later point. Note that this image does not visualize the concept of stateful property-based testing in general, but only the two depicted methods.

data insertion. However, the generation with the writer approach does not necessarily return all generated values. A generator of type `GenWithDBActions[A]` only returns a value of type `A`, although more data can be inserted into the database. If the generator is specifically implemented for the stateful property-based test, this is not a problem, as the type `A` can be chosen to contain all generated data. Alternatively, the type `A` can even be the abstract state itself. In that case, a generator of type `GenWithDBActions[State]` encapsulates both, the generation of the abstract state and the insertion of corresponding data into the database. When the reference graph generator is used, all generated values are contained in `A`, so the abstract state can be constructed. An alternative approach, which might seem plausible, is to first generate an abstract state and afterwards insert all data from the abstract state into the database. However, this can be difficult in practice, as in most cases the abstract state only contains a simplified representation of the data and not actual data that can be inserted into the database.

In this section, we assume that the reference generator approach is chosen, but the writer approach can be integrated in a similar fashion. There are different approaches to integrate the reference graph generation into the stateful property-based test. The initial state of the abstract model is generated by the `genInitialState` method. The `newSUT` method uses the generated initial state to create and prepare the system under test (SUT), which in our case means that data is inserted into the database. Figure 5.3 visualizes this relation between `genInitialState` and `newSUT`. As the content of the reference graph must be included in the abstract model and in the system under test, the reference graph must be accessible to the `genInitialState` and `newSUT` methods, respectively. The reference graph must be accessible to the `newSUT` and `genInitialState` methods, because they are responsible to include the content of the reference graph into the abstract model and into the system under test, respectively. We considered three approaches to integrate the reference graph generation.

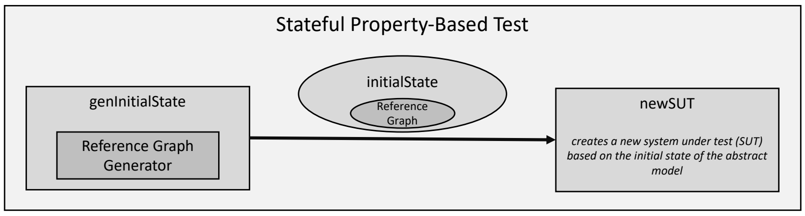
Stateful Property-Based Test
genInitialState
Reference Graph Generator
initialState
Reference Graph
newSUT
creates a new system under test (SUT) based on the initial state of the abstract model

Figure 5.4: Illustration of the *internal generation* approach. The reference graph is generated in the `genInitialState` method. The method appends this reference graph to the initial state so that the `newSUT` method can insert the content of the reference graph into the database.

5.5.1 Internal Generation

The first approach, which is illustrated in Figure 5.4, is to generate the setup in the `genInitialState` method. Conceptually, this makes a lot of sense, as the purpose of the reference graph generator is to generate the initial state of the application. A problem of this approach is that the `genInitialState` method only provides the state of the abstract model to the framework. The system under test is only created at a later point in the `newSUT` method. When the system under test is created in the `newSUT` method, the content of the reference graph has to be inserted into the database so that the database matches the abstract state. The abstract state is the only parameter of the `newSUT` method. Therefore, all required information to put the database into the desired state must be contained in the abstract state, which is passed to the `newSUT` method. This can be achieved by adding the generated reference graph to the abstract state. A drawback of this approach is that the reference graph has to be contained in the abstract state indefinitely, although it might serve no purpose for the abstract model of the application. We implemented a stateful property-based test using this approach as a proof of concept.

5.5.2 External Generation

In this approach, the reference graph is generated independently of the stateful property-based test. This approach is visualized in Figure 5.5. In order to make the reference graph accessible to the `genInitialState` and `newSUT` methods, it is passed to the test as an argument and made globally accessible inside the test. The advantage over the previous approach is that the reference graph does not need to be part of the abstract state anymore.

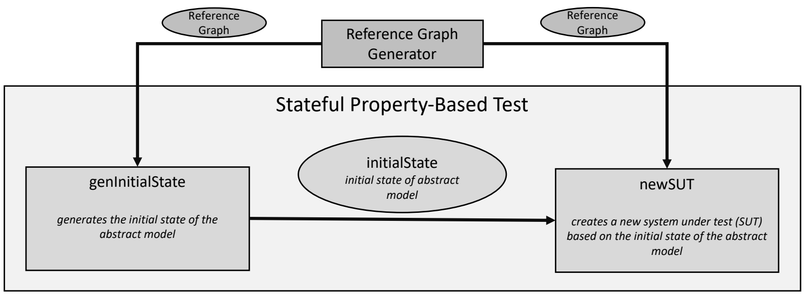
Reference Graph
Reference Graph Generator
Reference Graph
Stateful Property-Based Test
genInitialState
generates the initial state of the abstract model
initialState
initial state of abstract model
newSUT
creates a new system under test (SUT) based on the initial state of the abstract model

Figure 5.5: Illustration of the *external generation* approach. The reference graph is generated independently of the stateful property-based test and passed to the test as an argument.

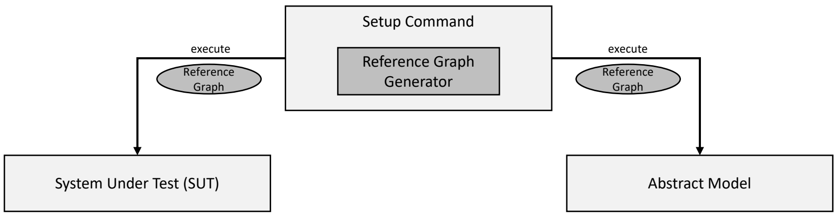

Figure 5.6: Illustration of the *generation in command* approach. An additional command called *Setup Command* is defined. The setup command can be executed on the system under test to insert the generated values into the database.

5.5.3 Generation in Command

The third approach is visualized in Figure 5.6. It is conceptually different to the previous two approaches, since the reference graph is not passed to the `genInitialState` and `newSUT` methods at all. Instead, an additional command called *Setup Command* is defined, which is responsible for the generation of the reference graph. When the command is executed on the system under test, the values are inserted into the database and when the command is executed on the abstract model, the values are added to the state of the model. The tester can use the part of the test which specifies the generation of the command sequence to define when the Setup Command can be executed. One obvious choice is to allow this command only as the head of the list, which corresponds to the behaviour of the previous approaches. However, it is possible to allow the insertion of more data at a later point as well. Here, the tester must be careful, as allowing multiple Setup Commands might lead to unreasonably long execution times. An advantage of this approach is that if a test fails, the Setup Command can be removed in the shrinking process just like any other command. This makes the minimal failing test case more comprehensible in situations, where the reference graph generation did not contribute to the failure of the test.

Chapter 6

Related Work

We viewed two different fields as related work: The first one is test data generation. There is a variety of different approaches for test case generation, including the generation of data specific for relational databases. We present some of these approaches in Section 6.1. The second related field consists of different approaches of property-based testing or testing in general in the context of stateful systems. We present these approaches and some case studies in Section 6.2.

6.1 Test Data Generation

Often, there are reasons why live data of a database cannot be used for testing:

- The database does not contain sufficient test cases.
- The database contains private or sensitive data.

There are approaches which use data mining tools like machine learning [Li, 2020] or statistical models [Patki et al., 2016] to examine the structure of real database content and generate synthetic database content with similar characteristics. As these approaches require real data to generate the test data, they are unsuitable for regression testing, as real user data might not exist yet. It can also be the case that existing user data is not diverse enough for some test cases, or there exists no data for a new feature that should be tested before production.

There are also specification languages like *SDDL* (Synthetic Data Description Language)[Hoag and Thompson, 2007] and *DGL* (Data Generation Language)[Bruno and Chaudhuri, 2005], which can be used to define how test data should be generated. They are meant for populating entire databases with realistic values, while we want to generate a small amount of custom data, which is suitable for a specific test case.

Some data generators focus especially on the performance and scalability of the generation [Gray et al., 1994, Rabl and Jacobsen, 2012, Alsharif et al., 2018]. Such generators are useful for benchmarks of database applications or

DBMS. However, as property-based tests are meant to be executed frequently, it is not feasible to generate a large database each time. Our focus is on generating diverse data, which covers as many corner cases as possible, instead.

Chays et al. [2000] design a database generator with the purpose of testing the correctness of an application. The generator takes the database schema as the input and generates content, which satisfies the specified integrity constraints. The generator assumes that all database tables are specified in a topological ordering, i.e. each table only references tables at a previous position. This assumption simplifies the generation of references, as the content for tables can be generated in the specified order. Whenever a referential integrity constraint references a column of a different table, a random value from that column can be picked, since content for the referenced table was generated previously. However, this implies that in contrast to our generator, the generator does not work on database schemas with referential cycles. In a later refinement, the generator automatically searches for a topological ordering, instead of expecting the tables to be topologically ordered in the database schema. But the problem of referential cycles is not addressed [Chays et al., 2004]. Houkjær et al. [2006] propose an alternative approach, which can handle referential cycles in the database schema. They build a graph model of the database schema, which is similar to our reference schema in the sense that each vertex represents a database table. Similarly to the other approach, the algorithm first generates content for tables that do not reference other tables. Data for a table with references can be generated, as soon as data for all tables it references was generated. However, when the algorithm finds a referential cycle in the graph, it proceeds as follows: The algorithm breaks the cycle by filling one of the foreign key columns with temporary values. Now, data for all tables in the cycle can be generated, successively. Once the generation is completed, the temporary values are replaced with actual references. Contrary to our approach, cycles on a row-level are not avoided.

The mentioned generators have the purpose of generating test case independent databases and generate all entries for a given table at once. Our approach on the other hand generates all entries separately in a demand-driven fashion. This is intentional to avoid the generation of data, which is not required for a specific test case. There are other approaches, where the generated database content is tailored for a specific test case. Mannila and Raiha [1985] propose a technique to generate database content based on a single SQL query using functional dependencies. Later, Binnig et al. [2007a] propose a technique called *reverse query processing*, which takes an SQL query and the corresponding query result as the input, and generates database content, such that the query can return the given result. *QAGen* is a similar approach but has the purpose to test a DBMS instead of a database application [Binnig et al., 2007b]. These approaches are substantially different from our approach, as they calculate a minimal database, which satisfies specific criteria, while our approach is largely randomized and takes a set of desired rows as input. To the best of our knowledge, no approaches of relational data generation specific for property-based testing were proposed at the time of writing.

6.2 Testing Stateful Systems

There already exists a lot of research in the context of stateful property-based testing. Arts et al. [2006] introduced the concept of stateful property-based testing in a tool called *Quviq QuickCheck*. It was used to test telecommunication software under development at Ericson. The approach was later extended to generate multiple command sequences and execute them in parallel, which is useful to test concurrent behaviour and thread-safety of applications. For this purpose, the tool *PULSE* was introduced, which gives the test a higher control over the schedule of an Erlang program and makes concurrent behaviour deterministically reproducible [Claessen et al., 2009]. Multiple race conditions were found using this approach in the *Mnesia* database management system [Hughes and Bolinder, 2011]. However, PULSE was not used in this case study due to technical difficulties. In a later case study, Quviq QuickCheck was used to test *AUTOSAR Basic Software* for Volvo Cars and concluded that it is more efficient and less costly compared to conventional testing approaches [Arts et al., 2015]. Another project tested using this approach is Dropbox [Hughes et al., 2016]. Examples of case studies for stateful property-based testing using the ScalaCheck framework are the e-commerce platform *Bizzkit* [Christensen et al., 2019] and the *Orchestra* system from *Qmatic* [Andersson and Lindbom, 2017].

Chapter 7

Conclusion

We presented an approach to extend the generators of property-based tests by including the information, how generated values can be inserted into a database. The approach is highly flexible and is especially useful to test custom scenarios, defined by the tester. Just like conventional generators, the extended generators are composable, such that a generator can be defined using existing generators. This is useful to avoid redundant code and simplifies the generation of values with numerous direct or indirect references.

Additionally, we proposed an algorithm, which automatically generates references for a set of desired database entries. This reference generator can handle situations, where it is difficult and laborious to manually write generators that do not leave out corner cases. We have proven that the generator is complete in the sense that all acyclic, connected database contents can be generated. Most existing generators generate all values for a given table at once, while our approach is demand-driven, as it proceeds on a row-level. The algorithm is implemented in a library such that it can be used to generate test data for existing projects. The implementation is integrated with the ScalaCheck library, as properties can be defined using the reference graph generator and the generator can be combined with other generators. All generated values can automatically be inserted into the database in an order, such that no referential integrity constraints are violated during the insertion.

Lastly, we proposed different techniques to combine the reference graph generator with the existing approach of stateful property-based testing. This hybrid approach has the advantage over a pure stateful property-based test that the database can be initialized with data, which cannot be created by any of the commands. This is a useful property when the stateful property-based test does not cover the entire application. Additionally, it can be useful to test, whether a new version of an application meets all expectations when it is executed on a database with legacy data. We implemented a test using the hybrid approach as a proof of concept.

7.1 Future Work

Shrinking is an important feature of property-based testing, as it makes test failures more comprehensible and easier to reproduce. In the writer approach, generators additionally collect database actions, which can be used to insert the generated data into a database. When shrinking is only applied to the generated values, but not to the list of database actions, then database content does not match the generated values anymore. Shrinking the generated values and the list of database actions in a way that they still correspond to each other is difficult to achieve in the writer approach. For this reason, we did not shrink the generated values. In order to improve the practicality of the writer approach, it makes sense to examine, if and how shrinking could be used in this case.

The contents of databases often have invariants between database tables, which cannot be represented by just referential integrity and *NOT NULL* constraints. Currently, the reference graph generator only supports these two constraints. This is not a problem for invariants, which only affect a single database row, as the tester can write custom generators for these rows, which take the invariants into account. But extending the reference graph generator to be able to handle more inter-table constraints would be a great improvement. One example is the ability to specify minimal and maximal cardinalities for relationships between the content of two database tables. The tester can currently specify that for a given database entry, at most n backward references are generated in the ExpandBackward method. However, it is still possible that the entry is referenced more than n times, as ExpandForward might create these references when other entries are visited. This problem can be solved by keeping track of how often each entry is being referenced and prohibiting the generation of further references once the maximal cardinality is reached.

The reference graph generator needs information about the used database schema, for example, which relationships between the tables exist. Currently, the tester has to manually specify these details. However, as the schema of a database can be queried, it is possible to automate this process. A suitable tool for this task is the code generator from the Slick library, as it is meant to generate code based on a database schema and is highly customizable[1]. When doing this, an approach should be chosen, where all generated code can be overwritten, so that the tester can customize the specifications and no flexibility is lost. This is important because the tester knows database invariants, which cannot be derived from the database schema. The code generator would significantly reduce the overhead of setting up the reference graph generator for a new project.

[1]`https://scala-slick.org/doc/3.3.3/code-generation.html`

Bibliography

A. Alsharif, G. M. Kapfhammer, and P. McMinn. Domino: Fast and effective test data generation for relational database schemas. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*, pages 12–22. IEEE, 2018.

D. Andersson and D. Lindbom. Generative scenario-based testing on a real-world system. Master's thesis, Chalmers University of Technology, 2017.

T. Arts and L. M. Castro. Model-based testing of data types with side effects. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, pages 30–38, 2011.

T. Arts, J. Hughes, J. Johansson, and U. T. Wiger. Testing telecoms software with quviq quickcheck. In M. Feeley and P. W. Trinder, editors, *Proceedings of the 2006 ACM SIGPLAN Workshop on Erlang, Portland, Oregon, USA, September 16, 2006*, pages 2–10. ACM, 2006. doi: 10.1145/1159789.1159792. URL `https://doi.org/10.1145/1159789.1159792`.

T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with quickcheck. In *Eighth IEEE International Conference on Software Testing, Verification and Validation, ICST 2015 Workshops, Graz, Austria, April 13-17, 2015*, pages 1–4. IEEE Computer Society, 2015. doi: 10.1109/ICSTW.2015.7107466. URL `https://doi.org/10.1109/ICSTW.2015.7107466`.

E. Axelsson. Compilation as a typed edsl-to-edsl transformation. *arXiv preprint arXiv:1603.08865*, 2016.

M. Azure. Finding circular foreign key references: Azure blog and updates: Microsoft azure. `https://azure.microsoft.com/en-gb/blog/finding-circular-foreign-key-references/`, Jul 2010.

C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 506–515. IEEE, 2007a.

C. Binnig, D. Kossmann, E. Lo, and M. T. Özsu. Qagen: generating query-aware test databases. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 341–352, 2007b.

N. C. Brown and A. T. Sampson. A trip down memory lane in haskell, 2009.

N. Bruno and S. Chaudhuri. Flexible database generators. In K. Böhm, C. S. Jensen, L. M. Haas, M. L. Kersten, P. Larson, and B. C. Ooi, editors, *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, pages 1097–1107. ACM, 2005. URL `http://www.vldb.org/archives/website/2005/program/paper/wed/p1097-bruno.pdf`.

D. Chays, S. Dan, P. G. Frankl, F. I. Vokolos, and E. J. Weyuker. A framework for testing database applications. In *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 147–157, 2000.

D. Chays, Y. Deng, P. G. Frankl, S. Dan, F. I. Vokolos, and E. J. Weyuker. An agenda for testing relational database applications. *Software Testing, verification and reliability*, 14(1):17–44, 2004.

L. Christensen, N. Heltner, A. Lascari, and N. Mølby. Sm2-tes project property-based testing of the bizzkit api. Accessed on 10.6.2021, 2019. URL `https://www.larspetri.dk/pdfs/tes.pdf`.

K. Claessen and J. Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In M. Odersky and P. Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000. doi: 10.1145/351240.351266. URL `https://doi.org/10.1145/351240.351266`.

K. Claessen, M. Palka, N. Smallbone, J. Hughes, H. Svensson, T. Arts, and U. Wiger. Finding race conditions in erlang with quickcheck and pulse. *ACM Sigplan Notices*, 44(9):149–160, 2009.

M. Grabmüller. Monad transformers step by step. *Draft paper, October*, 2006.

J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of the 1994 ACM SIGMOD international conference on Management of data*, pages 243–252, 1994.

F. Hebert. *Property-Based Testing with PropEr, Erlang, and Elixir: Find Bugs Before Your Users Do*. Pragmatic Bookshelf, 2019.

J. E. Hoag and C. W. Thompson. A parallel general-purpose synthetic data generator. *SIGMOD Rec.*, 36(1):19–24, 2007. doi: 10.1145/1276301.1276305. URL `https://doi.org/10.1145/1276301.1276305`.

K. Houkjær, K. Torp, and R. Wind. Simple and realistic data generation. In *Proceedings of the 32nd international conference on Very large data bases*, pages 1243–1246, 2006.

J. Hughes, B. C. Pierce, T. Arts, and U. Norell. Mysteries of dropbox: property-based testing of a distributed synchronization service. In *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 135–145. IEEE, 2016.

J. M. Hughes and H. Bolinder. Testing a database for race conditions with quickcheck: None. In *Proceedings of the 10th ACM SIGPLAN Workshop on Erlang*, pages 72–77, 2011.

W. Li. Supporting database constraints in synthetic data generation based on generative adversarial networks. In D. Maier, R. Pottinger, A. Doan, W. Tan, A. Alawini, and H. Q. Ngo, editors, *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, pages 2875–2877. ACM, 2020. doi: 10.1145/3318464.3384414. URL `https://doi.org/10.1145/3318464.3384414`.

S. Liang, P. Hudak, and M. Jones. Monad transformers and modular interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343, 1995.

I. Lightbend. Slick - functional relational mapping for scala. `https://scala-slick.org/`, 2012.

H. Mannila and K. J. Raiha. Test data for relational queries. In *Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems*, pages 217–223, 1985.

R. Nilsson. *ScalaCheck: The Definitive Guide*. Artima Press, 1 edition, 2014.

M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Maneth, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger. An overview of the scala programming language. 2004.

N. Patki, R. Wedge, and K. Veeramachaneni. The synthetic data vault. In *2016 IEEE International Conference on Data Science and Advanced Analytics, DSAA 2016, Montreal, QC, Canada, October 17-19, 2016*, pages 399–410. IEEE, 2016. doi: 10.1109/DSAA.2016.49. URL `https://doi.org/10.1109/DSAA.2016.49`.

T. Rabl and H.-A. Jacobsen. Big data generation. In *Specifying Big Data Benchmarks*, pages 20–27. Springer, 2012.

M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with spec explorer. In *Formal methods and testing*, pages 39–76. Springer, 2008.

Index