Event Execution Reproduction by Log Analysis

Master Thesis

Joanna Gasiewska
Department of Information and Computer Sciences, Utrecht University
ICA - 3188345

February 18, 2013
Abstract

Imagine an interactive application. On the top level it is driven by user events such as clicking button, requesting a service, etc. These events can trigger internal events such as function calls. Apart from that, an event may also cause a change in the state of the application. This may influence the way subsequent user events are executed. In case an error appears in one of them, it can be a result of a particular context in which the user event is executed and that context is the result of execution of previous user events. We assume the events are logged and our goal is to reproduce the error. First, we need to reduce logged sequence of events by eliminating the events that were not relevant towards the occurred error. In order not to spoil the production data, the reproduction is executed in a special test environment whose persistent state has the same structure as that of the production version, but is populated with different data. In our research we target these two issues by defining two terms: event commutativity and data similarity. We have implemented an algorithm that determines if events commute and an algorithm that finds similar data. We have also conducted an experiment that validates our approach.
Chapter 1

Problem statement

Imagine an interactive application. On the top level it is driven by user events such as clicking button, requesting a service, etc. These events can trigger internal events such as function calls. Apart from that, an event may also cause a change in the state of the application. This may influence the way subsequent user events are executed. In case an error appears in one of them, it can be a result of a particular context in which the user event is executed and that context is the result of the execution of previous user events.

These kind of errors are very difficult to find and eliminate in the testing phase since in event driven applications the number of possible user event sequences to test can be very large. There are techniques that target this problem \[27\] but as inherently in testing, they cannot guarantee detection of all errors. If we cannot eliminate such errors, then we must at least have a tool to diagnose them after they appear. Execution reproduction gives a good insight of what is causing the error, especially if accompanied with debugging or enhanced logging. However, its implementation is not an easy task. It brings forward the following difficulties:

- It is infeasible to record the full state of the application therefore at the moment of reproduction we do not know the exact original states of the application
- In order not to spoil the production data, the reproduction is executed in a special test environment whose persistent state has the same structure as that of the production version, but is populated with different data.

Therefore, it is in principle impossible to exactly reconstruct the same execution as in the original environment. Instead, we can construct an approximation that may be good enough to help analyze the cause of an error.

The assumption of the execution reproduction is that the sequence of the original user events is logged and the events themselves have been individually, and sufficiently well tested before deployment. When an error occurs, the likely cause of it is the execution of the previous events that had an effect on the erroneous event. Therefore, the first challenge of the reproduction is to find only those events that are relevant towards the error. This can be done by removing the so called commutative events.

Commutative property comes from algebra and is a property when changing the order of the operands, it does not change the result. Applying this to events, event $e_1$ commutes with event $e_2$ when execution of $e_1$ and then $e_2$ produces the same application state when executing first event $e_2$ and then event $e_1$. An example of such events: Copy and Cut whereas Copy and Paste are examples of commutative events.
CHAPTER 1. PROBLEM STATEMENT

Figure 1.1: Commutative events Copy and Cut. The result of executing them in any order is the same.

Figure 1.2: Not commutative events Copy and Paste. The result of executing them in different order gives different results.

The execution of Copy puts the text into clipboard and the execution of Cut does the same with addition of removing the text from the editor. Changing the order of those two events results in the same state which is the text in the clipboard and the editor with the text removed. However, Copy executed with Paste causes two lines to appear in the editor as the result. When the order of those events is reversed only one line appears.

To conclude, checking if events commute boils down to checking if the resulting state was changed after switching the order of the events. In practice this may be impossible to do, due to the difficulty of comparison of the application states. This applies especially to the applications using persistent state.
A different approach would be that a developer manually identifies for each event, which events are the commutative and which are not commutative. This is however hard to maintain if there are many types of events, and later when more event types are added. The commutativity of events could also be extracted by analyzing how the users used the software in the past. If many of them repeatedly used a combination of two events then it probably means that they do not commute. Such approach would require some amount of logged data about the application and does not guarantee that all relationships will be correctly identified. Therefore a method for determining an event’s commutativity is the first part of the research on execution reproduction.

**Definition 1** (Event commutativity problem). *Given two events \( e_1 \) and \( e_2 \) how to determine if \( e_1 \) and \( e_2 \) commute or not?*

After extracting the sequence of commutative events form the original execution, it needs to be fitted into the reproduction environment. This means that the original events need to be adjusted to the conditions in the reproduction server. The goal is to get the same results on the reproduction environment as on the original one. Events may take parameters so they may need to be replaced in the test environment. If we use the original parameters in the reproduction we may not get the same results, especially when the application uses a persistent state. For example if we have an event \( e_{\text{userExists}} \) that takes the parameter "\( joanna \)" and checks if a user named "\( joanna \)" exists in a database, this may have different results on the test environment than the original one. Different results can be expected because if the user does not exist in the test database, the execution path will differ. To reproduce the same execution path the parameter for the event \( e_{\text{userExists}} \) must be "adjusted". Such a parameter will be called *similar*.

**Definition 2** (Parameter similarity problem). *Which parameters should be chosen for the events so that their event execution gives the same results on the test environment?*

After solving the event commutativity and parameter similarity problems, execution reproduction can be made. If event \( e_x \) is the one we want to reproduce, we just need to filter commutative events from the events that proceeded \( e_x \) and find similar parameters for them on the test environment.
Chapter 2

Related work

There are a number of execution reproduction tools on the market. They all first need to record the data necessary for the reproduction. The difference is mainly in the kind of data and how it is collected. The data can be either system interactions [24], input parameters [14] or stack frame [5]. It can be recorded by instrumenting the source code or by modifying a compiler or an interpreter [5].

JRature [24] was developed to support operational testing of Java programs. This kind of testing involves ordinary users making first use of the software and the tool recording all their interactions. Then the trained testing personnel reviews the recording for further investigation. The testers can filter the recorded execution using the profiling component. The recording component modifies the Java API classes by adding special calls that catch all the interactions with the operating system. Each thread and all the objects it creates, along with the time duration of some operations such as querying file, network, and system time queries. Thanks to saving this information, it is possible to even reproduce the GUI visualization of the user interactions in the replay part.

ReCrash [14] is another tool for reproduction of Java programmes. In the monitoring phase it keeps track of all the methods and their parameters when they were on the call stack at any moment of the execution. The information is put on a special ReCrash stack and if a method finishes the execution without any exception then it is removed from the stack with all its method invocations. When the program crashes, the unit tests are created for each method from the stack with the parameters restored. ReCrash is lighter than JRapture because the amount of data recorded is smaller and the recorded information is stored in memory. The price of lightness is less precise reproduction; however, it is still able to successfully reproduce the set of crashes of programs with the inputs that caused them to crash.

Visual Studio 2010 (IDE for .Net) has a new feature in the Ultimate version which is capable of tracing and replaying execution. Depending on the nature of the project (web project, windows forms, ADO.NET), the events that interact with the framework are recorded (for example menu item selections, button clicks, Execute Reader or Execute Scalar). It is even able to backwardly debug a program when is set up to collect data about parameters and return values at all method entries and exits. This means, we can see what has previously occurred during execution, and most importantly, we can “step back in time” to see prior states of the application without having to restart the debugger (more information on the tool is available via MSDN link).

These tools are very accurate in the way the reproduction is done. They are capable of showing the exact execution steps that lead to the program crash. The overhead is minimized to make the programs still responsive. However, they do not address the problems of reproduction under a different environment. The major
obstacle is the persistence state on the reproduction environment that is different than on the original one. The solution presented in this thesis shows how we can deal with these things, though the resulting reproduction will be less accurate.
Chapter 3

Definitions

The kind of applications we consider here are event driven ones. This means that the flow of the application is determined by its events. Here is how we define an event:

**Definition 3 (Event).** An event $e$ is a function that may take parameters and changes the state of a program. In programming it could be implemented by a method call. If $S$ is the set of all possible application states and $P_e$ is the set of possible parameters for the event $e$, then we denote the type of $e$ as: $e : S \times P_e \rightarrow S$.

Our notion of an event is therefore broader than the definition in event-driven programming where an event is only initiated outside the scope of a program. It is however useful to make a distinction between an event that marks user interaction and an event that is just a consecutive internal method call. Therefore, we add two more event definitions:

**Definition 4 (User event).** A user event is an event that can only be triggered by a user.

Examples of user events are: clicking the Submit button on a WebPage, adding an item to a web shop cart etc.

**Definition 5 (Internal event).** An internal event is an event that is triggered by a user event or another internal event.

Internal events are mostly method calls that are initiated by a user event. Another example of an internal event is a system event of finished printing a document after the user event of pressing a print button.

We can treat event-driven application execution as a sequence of executed user events $[e_1(S_0, p_1), e_2(S_1, p_2), \ldots, e_n(S_{n-1}, p_n)]$. Each element in this sequence is an event that takes the state before the execution and a parameter. Events produce a state as their output which is an input for the next element in the sequence. For each event $e_k(S_{k-1}, p_k)$ in the sequence, $p_k$ is the event’s parameter and $S_{k-1}$ is its initial state. Executing event $e_k$ produces a new state $S_k$, which would be the initial state of the next event $e_{k+1}$ in a sequence.

For the events that do not take parameters the value of the second argument is $\emptyset$ and for the ones that take multiple parameters we treat them as a tuple representing one parameter.

Additionally to a resulting state, event execution can produce a log. This will be denoted using a function $\text{log}$. Hence, if the execution of an event $e$, that takes parameter $p$, on an initial state $S_0$ produces log $L$, this is denoted as $\text{log}(e(S_0, p)) = L$. The log after the execution of a user event sequence $e_1, e_2$ is the concatenation...
of the produced logs and is denoted as $\log([e_1(S_0,p_1), e_2(S_1,p_2)])$. Each user event triggers a number of internal events so in fact the log from a user event contains the sum of logs from all triggered internal events. The information we want to keep in the log is the control flow of the executed events. Control flow can be described using control flow points.

**Definition 6** (Control flow point). A control flow point is a program location immediately after a control flow statement, or the program location immediately after a statement that directly branches out of a control flow statement or the location of a method’s entry.

A control flow statement is a statement whose execution results in a choice being made as to which of two or more paths should be followed. For each of those paths, a control flow point will be inserted. Additionally, to deal with recursion, a control flow point is added at every method entry. Recursion can also change control flow and method entry is the only place to track it.

Control flow points are static points in a code that we can point out given a source code of events.

**Definition 7** (Execution path). An execution path is a sequence of control flow points that were passed during an execution. Execution path of a sequence of events $\pi$ is denoted as a function $xp(\pi)$.

Execution path is an essential concept in our solution. It still needs to be extended with a definition of how the execution paths can be compared. One of the research goals is in determining how close the paths are.

**Definition 8** (Closer execution path). Given two execution paths, path1 and path2, let $c(path1,path2)$ be a function that returns the length of the longest common subsequence of path1 and path2. $q2$ is a closer execution path to $p$ than $q1$ when $c(p,q1) < c(p,q2)$.

Longest common subsequence (LCS) is a problem, as the name suggests, where we want to find the longest common subsequence between two or more sequences. It should not be confused with the substrings. For example, having a sequence [A, G, C, A, T] and [G, A, C] there are three longest common subsequence of length two: [A, C], [G, C], [G, A]. LCS is also used often in implementation of file comparison programs such as diff.

A log contains dynamic information about taken execution paths. It is composed of a sequence of log entries.

**Definition 9** (Log entry). Log entry is a unit of a log and contains information about an executed control flow point.

A log entry corresponds to a dynamic control flow point. The correspondence between log entries and control flow points is injective. It means that every log entry must correspond to exactly one control flow point (in particular, it is not allowed to correspond to multiple control flow points). However, not all control flow points will produce a log entry. This can be because control flow did not reach them or the code could not be instrumented to log them (e.g. system libraries). We also assume that the order of the log entries reflects the order of execution of control flow points. Thanks to those constraints when reviewing a log we can check how the execution went even though the statements are in the source code.

In our solution we will compare the produced logs, therefore it is important to first define when the logs are equal:

**Definition 10** (Log equality). Two logs are equal when they have the same sequence of log entries.
In the text, for simplicity, we will use the term log as a bunch of physical log files created after execution of one user event. Therefore, when the logs for two events are compared, in fact one or more files are compared.

A log may contain repetitive information which is caused by loops or recursions in events. When comparing logs we are not interested in distinguishing if the same log entry was present 3 times or 5. Thus, we will pre-process the logs so that the multiple entries representing a loop (or recursion) will be reduced to just one. This is why we introduce a log that has the repetitive log entries removed.

**Definition 11** (Reduced log). *A reduced log is a log obtained from another log by removing duplicate segments of log entries whose length are at least 2.*
Chapter 4

Solution

Execution reproduction is typically carried out in two phases. First, execution is recorded and then it is replayed. Ideally, we want to record the full state of whole application after each program statement. However, this is not feasible. This would tremendously slow down an application, and logging full states of more complex applications that use databases would require an extremely large amount of disc space. Hence, in practice we log less information. However, the less information is saved, the less precise the replay is in the second phase of our reproduction.

In this solution, by removing commutative events and running the remaining events in the given sequence with similar (but not necessarily the same) parameters, we can obtain an approximated original execution on a different environment. The reproduction does not require the capturing of persistent state but is still able to give an indication of which event sequence and with what parameters this sequence leads to an error or some special state encountered on the original environment.

4.1 Logging

To accomplish any execution reproduction we need to first record an execution. As previously mentioned, it is better not to record occurred application states but rather to record execution paths by logging them. However, logging each executed program statement can have a big impact on the speed of an examined application and it can also take a lot of disk space. To reduce the number of log writes, we may save just the control flow information that indicates the order of executed program statements. Control flow is also a property that can be traced and reproduced on a different environment. The types of control flows are:

- basic control structures
- function call/return
- recursion
- exception
- function pointer/abstract method
- asynchronous function
- concurrency

For example for Java, the basic control structures are:

- decision statements (if-then, if-then-else, switch)
• looping statements (for, while, do-while)
• branching statements (break, continue, return)

In this thesis we will limit ourselves to applications which are not multi-threaded and work in a synchronous way and so we are not concerned with the last two kinds of control flow. With this assumption in mind, if we are able to reproduce the exact control flow, we should obtain the same execution path (logging will also be used in determining event commutativity that will be described later).

Each user event invokes a number of internal events. This means, to log one user event it is necessary to log control flow in a user event as well as control flow of all internal events that were executed afterwards. The way to implement this type of logging is by instrumenting the source code of all user and internal events. After each control flow statement an extra call to the logger is injected. For example, the if statement will be transformed in the following way:

```plaintext
if(x > 0)
{
  x++; // do the logging here
}
```

**Figure 4.1: Original code**

**Figure 4.2: Instrumented code**

When the modified code is now executed, every time the condition in the if statement is true, the if control flow will be logged. If nothing was logged this means the condition must had been false. Sometimes the if can have a condition that is a method call to a function returning boolean, for example:

```plaintext
if(isPositive(x) || isOdd(x))
{
  // do the logging here
  x++; // do the logging here
}
```

**Function isPositive and isOdd may also have some control flow that should be logged. In this case the logger should first record the control flow of the functions and then the if statement.**

Logging is done in control flow points. Each of these points should have a unique signature, so that later on it is possible to map those points to proper places in the source code. If we used the program written in Java, then the following information must be logged:

• package name
• class name
• method name
• control flow point type
• control flow point id

A control flow point id is an id that will allow to distinguish the same control flow point types (e.g: two if’s) in one method.

One user event can be executed multiple times. We need to be able to differentiate these calls. After each user event a special request id is generated and passed to
4.1. LOGGING

issued internal events. These request ids are added to a log file along with event’s control flow data. In this way it is also easy to see which internal events were issued by the user event.

In the case of user events, apart from control flow also parameter values must be logged. They are necessary for finding similar data. This is how the whole logging process looks like:

![Diagram](image)

**Figure 4.3:** User issues user event that causes generation of request id and logging of user event parameters and control flow of user and internal events

Special care must be taken with logging exceptions. We can divide exceptions into checked and unchecked. The first ones have to be handled in a code and they represent avoidable conditions that can be recovered. In Java, whenever such an exception occurs, control flow is transferred to a catch block where an exception is handled. In this cases a logger should be invoked to record the occurred exception. However, unchecked exceptions have no place to handle errors thus no place to log them. Those exceptions represent severe unexpected exceptional conditions (e.g: OutOfMemoryException) and they need not be handled in a code. This means they can appear anywhere in a code and they can change control flow, however we are not able to log this. For example:

```java
if(x > 0)
{
  // do the logging here
  x++;  
}
```

**Figure 4.4:** When x < 0 nothing is logged

```java
if(x > 0)
{
  // exception thrown here
  x++;  

  // do the logging here
  x++;  
}
```

**Figure 4.5:** Exception thrown before the if statement, so nothing is logged

We can see two different control flows in these examples but they will produce the same logs. To differentiate these two cases, a solution is to wrap a user event code, which is the most top level method in the application, with an extra try catch clause and add logging in a catch block just as for the checked exception. In this way we will log the occurrence of the exception which will make a difference in the logs produced by the examples above.
4.2 Event commutativity

Commutativity of events is useful in constructing effective test suites for event driven applications. Most of the time it is done by finding which events do not commute and therefore need to be tested together. They are commonly referred to as dependent or semantically interacting events in the literature.

In testing Ajax applications and in testing GUI, detection of semantically interacting events helped in creating test cases that were useful at detecting serious and relevant faults in applications [28] [16]. At the same time it reduced the number of generated test cases without losing the power of exposing errors. For example, having events: log in, press button, open window, close window we can construct $4^k$ test cases of length $k$. Some of them are less likely to reveal faults or make less sense to test. For example: close window, log in, press button, log in. Such a test will always fail because log in needs to precede all other events. By determining which events interact with each other, we can detect which events should be tested together in a test case.

4.2.1 Existing approaches

Here are some approaches for defining when events are semantically interacting.

Interprocedural Control dependence

Since our events are treated as procedures we can apply algorithms to check for interactions existing among procedures (interprocedural) [23] [19]. This is an extension of data flow analysis that goes beyond the method body.

The solution is based on a graph build from a program source code. Nodes are program statements or branch conditions and edges represent transfer of control. In this graph (let’s call it $G$) the following properties hold:

- A node $u$ postdominates a node $v$ if and only if every path from $v$ to the end node contains a node $u$.

- Node $u$ is control dependent on node $v$ if and only if $v$ has successors $v'$ and $v''$ such that $u$ postdominates $v'$ but $u$ does not postdominate $v''$.

Next, we extend a control flow graph with data dependencies. We add to each node a set of use variables (read value) and def variables (assign new value). Let $D$ be a function that takes a graph node and return its def variables; and let $U$ be a function that takes a graph node and returns a variable that is a use. Then the following occurs:

- Node $u$ is data dependent on node $v$ if and only if there exists a path $vWu$ in $G$ such that $(D(v) \cap U(u)) - D(W) \neq \emptyset$, where $D(W) = \bigcup_{n_i \in W(n_i \notin \{u, v\})} D(n_i)$.

- Node $u$ is syntactically dependent on node $v$ if and only if there is a sequence $n_1, n_2, \ldots, n_k$ of nodes, $k \geq 2$, such that $v = n_1, u = n_k$, and for $i = 1, 2, \ldots, k - 1$ either $n_i + 1$ is control dependent on $n_i$ or $n_{i+1}$ is data dependent on $n_i$.

The last definition is of syntactical dependence. Apart from it there is also a definition of semantic dependence. This kind of a dependence occurs when the semantics of a statement may affect the execution of another statement. A node $u$ is semantically dependent on a node $v$ if there are interpretations $I_1$ and $I_2$ of graph $G$ that differ
only in the function assigned to \( v \), such that for some input, the execution history of \( u \) induced by \( I_1 \) differs from that induced by \( I_2 \).

To find these dependencies interprocedurally, we use inlining. This is a replacement of nodes with statements where a procedure call is made with control flow graphs for the whole procedure. Inlining has disadvantages as taking a lot of memory, or scoping of local variables because the whole module is viewed as one big procedure [11]. Furthermore, it is impossible to represent recursive procedures. The overall technique does not handle reference parameters, global variables and recursion for direct and indirect data dependencies. Direct dependencies exist when either a definition of an argument (actual value passed) in one procedure reaches a use of the corresponding parameter (parameter in the function definition) in a called procedure or a definition of a parameter in a called procedure reaches a use of the corresponding argument in the calling procedure.

**Common variables and exchanged messages**

The reason why events are interacting is because they share variables or exchange messages [21]. This can be determined by analyzing a source code of event handlers by observing common variables reads and writes.

The algorithm in [21] is based on four cases that define the way in which variable states are shared in event handlers. Each case produces a test case with a sequence of interacting events (not necessarily two events). Let’s define \( e_{Def} \) as an event in which an event handler assigns a value to a variable \( v \) and event \( e_{Use} \) as an event that makes a reference to a variable \( v \). Then we can build the following test cases:

- \([e_{Def}; e_{Use}] - one event assigns \( v \) and the next event references it.
- \([e_{Def}; e_{Def} \& e_{Use} \& e_{Use}] - one event assigns \( v \) and there are two consecutive events in which one assigns and references \( v \) and the other event only references \( v \).
- \([e_{Def}; e_{Use} \& e_{Use}] - one event assigns \( v \) and the next one references it and assigns a different variable \( v' \) and then, the next event first references \( v' \) and then assigns \( v \).
- \([e_{Def}; e_{Def}; e_{Use} \& e_{Use}'] \& [e_{Def}; e_{Def}; e_{Use} \& e_{Use}'] - the first two events assign values to different variables \( v \) and \( v' \) and the next event references both of the variables.

The number of test cases created in this way was compared with the number of test cases generated by a permutation and combination. The number of created test cases turned out to be 10 times smaller. Though less test cases, the test suite remained effective in exposing errors. This solution looks like a data flow analysis.

We divide data operations into data definitions (data creation, initialization) or data use. Then, by tracking the def-use associations we can find how data flows. The associations are between memory location such that there is a control flow path between them and no intermediate redefinition or undefinition of the location [13].

This approach has multiple drawbacks. Static analysis has some limitations such as ignoring the calling context and intra procedural flow of control [22]. In OO world it is because of presence of virtual functions, reflection or multi-threading. Event handlers may exchange state information without having common variables as in the case of these two events:

```javascript
event e1()
{
    var dbConn = "database connection string";
}
Database db = new Database(dbConn);
db.update("INSERT Customer ("Joanna", "Gasiewska")");
}

event e2()
{
    var dbConn = "database connection string";
    Database db = new Database(dbConn);
    db.update("DELETE Customer ("Joanna", "Gasiewska")");
}

These two events exchange state of the Customer tables, however from the static analysis it would be very hard to come up to such a conclusion.

Run-time state

Paper [28] describes how run-time state of widgets in a GUI can be used to identify events that interact. This method was invented for the purpose of testing GUI. Each event produces a state on a GUI by changing the properties of its widgets. If a sequence of events $e_1, e_2$ produces a state that is "different" than the state when $e_1$ and $e_2$ are executed in isolation then $e_1$ and $e_2$ are interacting events as shown on figure 1.2. The states are "different" when they fall into one of the cases. $e(S)$ denotes a state after the execution of event $e$ on state $S$

- there exists a widget property whose value is not changed after $e_1(S_0)$ or after $e_2(S_0)$ but it is changed after $e_2(e_1(S_0))$
- there exists a widget property whose value is changed after $e_1(S_0)$ or after $e_2(S_0)$ or by both and it is changed to another value after $e_2(e_1(S_0))$
- there exists a widget that is created after $e_1(S_0)$ or after $e_2(S_0)$ and it is created after $e_2(e_1(S_0))$ but with a different value
- there exists a widget that was disabled in $S_0$ and became enabled after $e_1(S_0)$ and then event $e_2$ is executed on this widget

Interacting events are used to build a new model of GUI, based on which test cases are generated. This model proved to identify complex relationships among GUI event handlers that were causing serious failures.

This solution for interacting events is meant to work for GUI events and would be hard to apply to other kind of events. However, the idea of comparing of a state after executing events in a sequence with the state after executing them in isolation is interesting and could be applied to a bigger range of applications. Its implementation may however be difficult. Executing an event in isolation requires either having a copy of the environment with the same state for each event or rolling the state back after an event execution. This may not be physically possible for complex applications using a persistent state. Moreover, the comparison of a final state can be elaborate for such applications.

Abstract state

It is a combined approach of static analysis, run-time state and manual input. It was used for testing of Ajax (Asynchronous Javascript And XML) applications [16] which is a bundle of technologies combined to build faster and more responsive web sites. DOM (Document Object Model) keeps the state of how a web page should be displayed and the XMLHttpRequest is responsible for communicating with a web server. These calls between the client and the server are done asynchronously and
this creates difficulties for testing. The paper [16] describes how to automatically generate test cases for Ajax applications and limit their amount by the means of semantically interacting event detection.

First, a set of methods that affect a DOM is identified using static code analysis. Then these methods (events) are traced by logging the DOM state changes. The number of all possible states is unbounded, therefore only an abstraction is used. For example a text field state can be abstracted to having empty or non-empty string instead of a particular value. When creating test cases, these states are used to create FSM. Events are the transitions between the states. Some states and transitions in FSM can be undetected, this is why some manual input may be required. Next, we can generate the test cases from the built FSM. Instead of generating all possible sequences, only the ones that have semantically interacting events are used. The two events $e_1$ and $e_2$ are said to be semantically interacting if there exists a state $S_0$ such that their execution in $S_0$ does not commute. That means if $e_2(e_1(S_0))$ is a state created after execution of a sequence of $e_1, e_2$ on state $S_0$ and $e_1(e_2(S_0))$ is a state created after execution of a sequence of $e_2, e_1$ on state $S_0$, then if these two resulting states are different this means that $e_1$ and $e_2$ interact.

This method successfully detected the injected faults in a tested program and reduced the number of test cases by around 80%. The idea of commuting events is interesting and applicable to a wider range of programs (not only Ajax ones). The only problem arises when comparing a state for applications that have no output. After abstracting the DOM we can have a finite number of states and we can build FSM. For applications without output we would need to use the states of all variables which can be unbounded.

**4.2.2 Commutative events in execution reproduction**

By taking a closer look at the described approaches, it becomes evident that, they have the same notion of what dependent (semantically interacting) events are. The first two approaches are static and the last two are dynamic. We decided to use the dynamic approach in our solution, because the static one is more exploited and has limitations described before 4.2.1

**Definition 12** (Event dependency definition). Let’s take two user events: $e_1, e_2$; $e_1$ depends on $e_2$ if an execution of $e_1$ affects an execution of $e_2$.

We need to be more precise here in what 'affecting' means. In the mentioned approaches [28] [16] it was defined in terms of a change in the resulting GUI state. If one event depends on another, then executing them in a sequence will result in a different GUI state than what we would get by 'merging' the GUI states that result form executing those events separately, as in the Figure [12]. Either the state of widget properties was changed [28] or an abstracted DOM [16] was different.

The described approaches used detection of dependent events for testing optimization. Dependent events have to be executed in a specific order, thus they are non-commutative. In the context of application testing the goal is to make the generation of test cases more effective by generating only sequences of non-commutative events so they can expose the errors that result form the interaction between the events. In the case of execution reproduction, the goal is to remove events which are not likely to contribute to the observed error. We want to reproduce only the sequence of non-commutative events by removing the commutative ones. Therefore, our goal is to define when the events commute.
Definition

As outlined in section 1, to check event commutativity we would need to compare the states of an application after changing the order of the execution of events. Comparing all application states is infeasible, so we need a way of abstracting them. There is a relation between application states and execution flows. Application states change when an event is executed due to the event executions which in turn induces a 'flow'. We can define 'flow' as data flow, or alternatively as control flow.

Data flow is based on control flow and it is used to investigate how values of data that are associated with variables can affect the execution of programs. It is mostly used with static analysis. The run-time state approach can be seen as a dynamic version of data flow analysis. In that approach the properties of widgets were compared in run-time. For our solution this is not applicable because our events do not necessarily make changes on GUIs but any on kind of variables including the ones representing persistence. We decided to track application state changes using control flow.

Control flow refers to the order in which program statements are executed. Control flow statements are decision points where a decision for which statements should be executed next are made. The reasons for changes in control flow of a deterministic, single threaded, synchronous application can be: a parameter change, an application state change, or an exception.

If we execute an event twice with the same parameters, the resulting behavior can still be different if it is executed on different initial states. This usually (but not always) shows up in different control flows. Therefore, control flow will be used as an abstraction of the application state. Because frequently in literature, control flow is used in a static analysis as a hypothetical path the program may take, we introduce the term execution path as the actual path that was logged after the execution. We will use it to formulate the definition of commutative events. Let us first define event dependency using control flow.

Definition 12a (Dependent events). Let $xp$ be a function that extracts the execution path from a given sequence of events. Event $e_1$ depends on event $e_2$ if the parameters $p_1$ and $p_2$, an initial state $S_0$ and intermediate states $S_1$ exist $S_1'$ after the execution of a sequence $[e_1(S_0, p_1), e_2(S_1, p_2)]$ and sequence $[e_2(S_0, p_2), e_1(S_1', p_1)]$, we have $xp([e_1(S_0, p_1)]) \neq xp([e_1(S_1', p_1)])$.

Dependent events are also non-commutative. If two events are dependent, then executing them in reverse order causes different execution paths. However, non-commutative events are not necessarily dependent.

\[ e_1 \text{ depends on } e_2 \Rightarrow \neg \text{com}(e_1, e_2) \quad (4.1) \]

But:

\[ e_1 \text{ depends on } e_2 \not\Rightarrow \neg \text{com}(e_1, e_2) \]

Let’s illustrate this by the means of an example. If we have the following two events $e_1$ and $e_2$: 

\[ \]
4.2. EVENT COMMUTATIVITY

var global = 0;
event e1()
{
    if(global > 0)
        do something
}
event e2(int value)
{
    global = global + value;
}

No matter on which state we execute \( e_1 \), it will not affect (the execution path of) \( e_2 \). Conversely, \( e_2 \) has an effect on \( e_1 \), because it influences whether the if will get evaluated to true or false. Thus, dependency and non-commutativity cannot be equivalent. Instead, we have the following equivalence:

\[
\neg \text{com}(e_1, e_2) \Leftrightarrow e_1 \text{ depends on } e_2 \lor e_2 \text{ depends on } e_1 \tag{4.2}
\]

Thus, in order to determine if two events are non-commutative, it is enough to show that the event pair is dependent in at least one direction. From that conclusion we can define commutative events as:

**Definition 13** (Commutative events). Let \( xp \) be a function that extracts the execution path from a given sequence of events. Event \( e_1 \) commutes with event \( e_2 \) if for all parameters \( p_1 \) and \( p_2 \), initial state \( S_0 \) and intermediate states \( S_1 \) and \( S_1' \) and for all sequences \( seq_1 \) and its reverse \( seq_2 \), such that \( seq_1 = [e_1(S_0, p_1), e_2(S_1, p_2)] \) and \( seq_2 = [e_2(S_0, p_2), e_1(S_1', p_1)] \), we have \( xp(seq_1) = xp(seq_2) \). The property will be denoted as \( \text{com}(e_1, e_2) \).

According to the definition of an execution path it is a sequence of control flow points. Thus, paths are the same when sequences are the same. As previously mentioned, execution paths are stored in logs, so in fact, we are going to compare the log entries corresponding to the control flow points in the implementation of commutativity checking.

**Initial state and input parameters**

We can influence the results of the checking of event commutativity by the choice of an initial state. Let’s assume the parameter \( value \) in the event \( e_2 \) of the previous example is always non-negative. We will modify event \( e_1 \) a bit.

```plaintext
event e1(var x)
{
    if(global + x > 3)
        do something
}
```

If we execute \( e_1 \) and \( e_2 \) in a starting state \( S_0 \) such that \( S_0 = \{global = 200\} \), then the if branch in \( e_1 \) will always evaluate to true, this means the events commute. However, when we set \( S_0 \) to be \( S_0 = \{global = 0\} \) the if branch will evaluate to false in some cases of \( x \) and when running in \( S_0 \), but after execution of event \( e_1 \) it evaluates to true in some cases of \( x \), meaning the events do not commute.

Finding one case where events do not commute is enough to conclude that the events do not commute in general. It is important to try testing commutativity using different initial states in order to find such a case.

A similar issue occurs with the input parameters for the events. Using the previous example of events, let’s assume the initial state \( S_0 \) is \( S_0 = \{global = 0\} \). Running the tests for dependency we could pick the following test cases:
Running these two tests we explored all possible execution paths (if evaluated to true and false) and as a result we see the execution paths never changed after reversing the order. That would imply the event e1 and e2 commute. This is however not true because we do have a test case that will prove the events do not commute. An example of such a test case is:

<table>
<thead>
<tr>
<th>test sequence</th>
<th>execution path</th>
<th>reversed sequence</th>
<th>execution path</th>
</tr>
</thead>
<tbody>
<tr>
<td>e2(0), e1(4)</td>
<td>if true in e1</td>
<td>e1(4), e2(0)</td>
<td>if true in e1</td>
</tr>
<tr>
<td>e2(1), e1(1)</td>
<td>if false in e1</td>
<td>e1(1), e2(1)</td>
<td>if false in e1</td>
</tr>
</tbody>
</table>

The conclusion here is the same: we need to test the events using different sets of parameters. As such, in the definition of commutative events we ought to use all possible input parameters for the two events. In practice this is infeasible. Therefore, we need to find a criteria for the choice of initial state and input parameters that will allow the creation of these special cases.

Since our definition of commutativity depends on execution paths, a good criteria would be a path coverage. If events do not commute then there exists an execution path that is different in a test sequence than in the reversed order. The more paths are covered, the more chances of finding such a path. The full path coverage does not however always guarantee that the events that appear to be commutative are in fact not commutative. If we look at the first table with the inputs for the events of the previous example, e2 had one path and e1 had two paths and all of them were covered. Still we managed to get a third test case (second table) that shows the events do not commute. Because the solution is an approximation, we treat it as a possible error.

**State resetting**

In the two event sequences from the commutative events definition we use the same input parameters. It is done in order to exclude parameter change form the factors influencing the control flow. For the same reason, the two event sequences have to be executed using the same initial state. Here is an example that motivates this argument:

```javascript
var global = 0;

event e1()
{
    if(global > 0)
    {
        do something
        global++;
    }
}

event e2(int x)
{
    return x + 3;
}
```

Here the state changes with the state of a variable global so we have $S_0 = \{\text{global} = 0\}$ initially. After executing $[e1(S_0, \emptyset), e2(\{\text{global} = 1\}, 5)]$ the state is changed to $S_1 = \{\text{global} = 1\}$ and if we do not reset it the reverse execution $[e2(S_1, 6), e1(S_1, \emptyset)]$ would imply that the events do not commute, but in fact they do.

Resetting a state means that if we use a persistent state in an application, it has to be rolled back. This is why checking for commutativity should be executed in an isolated test environment where the events are executed only by the testers and there is no risk of harming the data.
4.2. EVENT COMMUTATIVITY

Black-box components

The programs that we consider for an execution reproduction are deterministic. That means the same execution path gives the same results. As results we do not mean an output, but rather a behavior. Let’s imagine a console application that prints a number that a user provides. The output changes depending on what number was provided but the behavior stays the same - the number is printed out on a console. In the execution reproduction of such an application we are more interested in obtaining the occurred behavior, rather than the exact output.

Sometimes a part of a behavior of a program is obscured from a programmer. The programmes using libraries, frameworks or web service hide the computational details from a developer. Therefore, we are limited in how we can verify if a behavior was changed or not. We can only judge whether something suspicious happened in the execution by looking at the output. These sort of programs create a limitation in terms of execution reproduction. This also applies to determining whether or not an event is commutative using our definition. For example:

```javascript
var currentMode = paint;

// setDrawMode
event setDrawMode() {
  currentMode = draw;
  Console.draw();
}
```

These events should not commute because `setDrawMode` should be called before the `draw` event. The `draw` event calls the internal event `Console.draw()`, but there is no way to check what is the execution path in that event because it is a call to an external library. The execution paths derived after executing:

\[
\begin{align*}
  \text{[setDrawMode}\{\{\text{currentMode }= \text{ paint}\}, \emptyset\}, \text{draw}\{\{\text{currentMode }= \text{ draw}\}] \\
\text{[draw}\{\{\text{currentMode }= \text{ paint}\}, \emptyset\}, \text{setDrawMode}\{\{\text{currentMode }= \text{ paint}\}] 
\end{align*}
\]

are the same. So, the events will commute according to 13 but in fact they should not.

Singleton pattern

Conversely, some events can appear to commute because of certain programming patterns. This applies to a singleton design pattern. In this pattern, the goal is to have only one instance of an object during the runtime of a program. The way this is implemented is by creating a class with a method that creates a new instance of the class if one does not exist. If an instance already exists, it simply returns a reference to that object. Let’s analyze the following two events:

```javascript
// getCanv1
event getCanv1() {
  var canvas = Canvas.getInstance();
  countElements(canvas);
}

// getCanv2
event getCanv2() {
  var canvas = Canvas.getInstance();
  countElements(canvas);
}
```

These events do exactly the same thing that do not affect each other which is getting the number of elements in the canvas. However, the canvas itself is implemented using singleton pattern in the following way:
public class Canvas {
    private static Canvas instance = null;

    private Canvas() { }

    public static Canvas getInstance() {
        if (instance == null) {
            instance = new Canvas();
        }
        return instance;
    }
}

When we apply our test for commutativity by first executing
\[
\text{getCanv1\{instance = null\}, getCanv2\{instance != null\}}\]
and then \[
\text{getCanv2\{instance = null\}, getCanv1\{instance != null\}}\]
we can see that the execution path of the getCanv1 event in the second sequence
is different than the first one because in the first sequence getCanv1 has to create
an instance of Canvas whereas in the second sequence this will not happen. This
makes the events getCanv1 and getCanv2 to be classified as non-commutating.
However, this does not mean this is a wrong result. The execution of the first event
in a sequence affects the next one because the latter does not need to instantiate
Canvas. This is thus, a result that comes from a kind of coding style, and not the
program's semantics. Given the possibility of this result, this is something we might
want to avoid.

Exceptions due to external circumstances

As previously mentioned, among the three things that can influence control flow,
one of them is an exception. The reason for occurrence of an exception can be from
some external conditions. This can be for example: file was removed, connection was
broken, memory was exhausted. If we execute events in a sequence and an exception
occurs in one of the steps in the sequence and then we execute the sequence in the
reverse order but this time no exception occurs, we will conclude that the events do
not commute. However, the reason for the exception was not the semantics of the
events but rather external circumstances in which the events had been executed.

There is no way of eliminating such external circumstances, we can only reduce
the chance of exception occurrence by thoroughly testing the application in advance.

4.2.3 The algorithm

For the purpose of an execution reproduction, an input program (a program we try
to reproduce) is instrumented to log the control flow points encountered in run-
time. This gives us information about the execution path in a way described in \[4.1\]
We will use the produced log to compare the execution paths of an input program
events.

Here is the algorithm for checking if event $e_1$ commutes with event $e_2$:
4.2. EVENT COMMUTATIVITY

Algorithm 1 Algorithm determining if event $e_1$ and $e_2$ commute

1: procedure ARECOMMUTATIVE($e_1$, $e_2$)
2:     e1CoveredPaths := {}
3:     e2CoveredPath := {}
4:     while e1CoveredPaths $\neq$ getAllPaths($e_1$) $\lor$ e2CoveredPaths $\neq$ getAllPaths($e_2$) do
5:         resetState()
6:         test := createTestCase($e_1$, $e_2$)
7:         testLog := execute(test)
8:         e1CoveredPaths := e1CoveredPaths $\cup$ {getCoveredPath(test, $e_1$)}
9:         e2CoveredPaths := e2CoveredPaths $\cup$ {getCoveredPath(test, $e_2$)}
10:        resetState()
11:        reversedTest := reverseTestCase(test)
12:        reversedTestLog := execute(reversedTest)
13:        if testLog $\neq$ reversedTestLog then
14:            return false
15:        end if
16:     end while
17:     return true
18: end procedure

The main loop is executed until events $e_1$ and $e_2$ have all paths covered. In the loop body, test cases are created after the initial state is reset with a call to resetState(). A test is created using createTestCase and it has to fulfill the following conditions:

- it must contain consecutive calls to $e_1$ and $e_2$
- the parameters for $e_1$ and $e_2$ should be chosen such that they maximize their coverage

Next, the generated test case is executed and it produces a log testLog. Using this log we may check which execution paths were covered by each event and mark those paths as covered. In the following step, the state is set back to the one that was before the test execution (line 7). This step is important for the reasons presented in 4.2.2. Then, we obtain the reversed test case by calling reverseTestCase. This test case should fulfill the following conditions:

- it must contain the consecutive calls to $e_2$ and $e_1$
- the same parameters are used for the events as in the original test

The reversed test case is executed and produces a log reversedTestLog. Because the execution of $e_1$ and $e_2$ was reversed, this log has its log entries reversed as well. We need to put them in a right order before a comparison is made. The logs are compared according to definition 3. If the logs are not equal, false is returned which means that the events do not commute. Otherwise, if all test cases were exhausted and the compared logs were always equal then we return true which means that the events do commute.

The condition in the while loop may be very hard to achieve. It requires 100% path coverage. The number of required test cases that need to be generated is related to the cyclomatic complexity of the events. The higher the complexity, the more difficult it gets to generate a test for a specific path. Therefore, the loop condition could be programmed to return true when the coverage is not necessarily 100% but some fixed minimum bound.
There are two ways of increasing path coverage, the first is by setting different initial states, and the second is by changing the parameters. Choosing parameters to obtain a specific execution path is not easy and is a topic explained in the data similarity section 4.2.3.

The commutativity algorithm 4.2.3 is executed for all events that we try to reproduce. We test pairs of all events on the test environment and we update commutativity matrix with all relations for the events. A value of 1 in the matrix means the events commute, and a value of 0 means they do not commute. As an extra information we also add the path coverage with which the value was calculated, since it is not always 100%. Here is an example of the matrix output:

<table>
<thead>
<tr>
<th></th>
<th>LogIn 0/70.0</th>
<th>AddOrder 1/80.0</th>
<th>AddProduct 0/75.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>LogIn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>AddOrder</td>
<td>0/60.0</td>
<td>1/63.0</td>
<td>0/70.0</td>
</tr>
<tr>
<td>AddProduct</td>
<td>0/65.0</td>
<td>0/70.0</td>
<td>0/60.0</td>
</tr>
</tbody>
</table>

This matrix should be read as the event in the column commutes with the event in the row. The number after / is the coverage value in percent.

4.3 Data similarity

According to the definition of an event, it may take parameters. Similar data are a group of parameters that we can use to replace the original parameters in an event call, and it will cause the same behavior of the event. A behavior of an event executed with different parameters is the same when the execution path is preserved.

The problem of finding similar data is related to the path coverage problem in testing. Testers try to build tests that will cover as many execution paths as possible. Since it is impossible to cover all execution paths, one way to achieve that is by varying the parameters of methods such that it will cause different execution paths. When that process has to be automated, another problem arises: how to find parameters that would lead to a specific path. This is an old problem, that has existed for around 40 years and numerous solutions have been proposed. We describe some of them in the following section.

4.3.1 Existing approaches

We can divide the approaches of automatic test input generators into two groups: random and structural. Random generators select the inputs randomly from some distribution [10]. Structural generators require internal knowledge about the program, such as the source code. These generators can be built using symbolic execution, or search based algorithm [7]. Since we assume that the source code of the reproduced program is available, we are going to concentrate only on structural approaches.

Symbolic execution

Symbolic execution is an execution of a program using symbolic instead of concrete values for variables. Input data as well as the output is represented using formulas. Symbolic execution is used in combination with a constraint solver to generate test input data that will lead to different execution paths.

Paper [30] uses symbolic execution to find inputs satisfying all possible execution paths. First, an EFSM (extended FSM) is built from a program where a state transition is a tuple of a current state, the next state, the predicate and the action (set
of assignments that give values to variables). Then, we can change the current state into the next one, if the predicate evaluates to \textit{true}. When making the transformation, the variables are updated by performing the action of the transformation. We traverse the graph starting from an initial state to a terminal one by selecting the transitions whose predicates satisfy the path condition by applying the actions of the transformations. The path condition is a collection of predicates that lead to a current state along the current path. If we cannot find a satisfying transition, then we backtrack. When we reach the terminal state, the constraint solver resolves the current path condition.

Symbolic execution has an inherent difficulty in handling arrays, references, loops and recursions. Some of the constraints may be very difficult to compute, especially when there are many indirect relationships between program conditions. One way to deal with these difficulties is described in [26], where symbolic execution is incorporated with heuristics (so called dynamic symbolic execution). This was used for automatic test case generation for a Microsoft testing tool - Pex. Along with the symbolic execution where the path constraints on the inputs are collected, the program is executed with concrete inputs in parallel. Each explored path (with concrete values) has a fitness value assigned. The value indicates how close the path is in covering the test target. Another measure is a fitness gain that reflects whether the choice of the branch in the past helped to increase the fitness value or not. This gives a possibility of a guided choice between branches which can be very helpful, especially if we have loops with conditions in a loop body. The decision which branches to visit inside the loop can be problematic for a constraint solver. The information about the previously executed paths helps to make this decision.

Chaining approach

In this approach, data flow is used to guide the selection of a branch of control flow [9]. First, a graph is created from all statements in a program. Each node represents a single statement and has a set of variable definitions and variable uses bound to a node. Then the program is executed with the aim of reaching the selected node. During the execution, control flow is monitored and choices must be made about which branches should be taken in order to reach the target node. The flow is influenced by changing the input that is generated using a function minimization search algorithm on the branch condition. If no proper input can be found, then the chaining approach tries to alter the flow by giving priority to the nodes that have definitions of variables used in the problem node.

This approach belongs to the goal-oriented data generators. Contrary to the path-oriented generators, the aim is to execute a program statement irrespective of the taken path. With that assumption it is hard to predict the coverage given a set of goals. For the purpose of the execution reproduction, we are also interested in achieving a specific path, rather than a program statement.

Genetic algorithm

Genetic algorithm is a meta heuristic search that was inspired by the process of natural evolution. It begins with a random population of solutions called chromosomes. For each chromosome a value is calculated that indicates how good a chromosome solves the problem - fitness score. Then two members of the population are selected, the chance of being selected is proportional to a chromosome’s fitness score. The pairs of chromosomes are crossed over (recombined) and mutated and then added to the new generation. The process is repeated for new generations until a solution is found, or a maximum number of generations is reached. Fitness scores guide the evolution towards the optimal solution and bias selection ensures that we do not
run into a local maximum.

Genetic algorithm can be used both as the path-oriented or goal-oriented solutions to test input data generation. A tool called TGen was build by applying the goal-oriented algorithm [18]. As previously mentioned, we are more interested in the path-oriented approach in our solution. This approach typically involves the construction of a control flow graph, the selection of a path and the generation of test data for that path. This is how the automatic test data generator for path testing in [15] was implemented.

In the constructed control flow graph of a program each branch is denoted by a label and a path is a combination of those labels. A path is selected as the target path, the test cases (the chromosomes) are generated until there is a test case that can be executed along the target path. The first generation of the test cases is generated randomly. The survivors of the generation are chosen according to the fitness function which in this case is a value that indicates the distance between the output path of the test case and the target path. The surviving test cases are crossed over to create a new generation of chromosomes and mutated with some probability. The crucial part of the algorithm is a calculation of the fitness score. In the described paper [15] the authors use an extended Hamming distance between the paths.

Standard Hamming distance measures the number of differences between two strings or in other words it is the number of substitutions that have to be done to change one string to another. For example, having two strings: "10011" and "11010", the Hamming distance is equal to two because the strings differ in position 2 and 4. The branches of the tested program are labeled, so the path is identified using a string composed of those labels. We could use the Hamming distance of those strings as a measure of how different the paths are. However, the requirement of the Hamming distance calculation is that the strings must have the same length which does not necessarily happen with the program paths. Because of this an extended version of the Hamming distance was introduced.

In the extended version the paths are represented as the sets of subsets of consecutive branches of fixed length. For example the path \{a, b, c, d, e\} after division to subsets of length two will be represented by \{\{a, b\}, \{b, c\}, \{c, d\}, \{d, e\}\}, this is called a second order set. A third order set is then \{\{a, b, c\}, \{b, c, d\}, \{c, d, e\}\}. Starting with the first order set, the comparison is made between the paths using their order set representations. The distance \(D_n^{i,j}\) between the path \(i\) and \(j\) in the order \(n\) is calculated in the following way:

\[
D_n^{i,j} = |S^n_i \oplus S^n_j|
\]

\(S^n_i\) represents the \(n\)-th order set of path \(i\) and \(S^n_j\) represents the \(n\)-th order set of path \(j\). The operation \(\oplus\) denotes symmetric difference. For sets \(A\) and \(B\) this is equivalent to:

\[
A \oplus B = (A \cup B) - (A \cap B)
\]

Path similarity \(M\) using \(n\) ordered sets of paths \(i\) and \(j\) is then formed as:

\[
M_n^{i,j} = 1 - \frac{D_n^{i,j}}{|S^n_i \cup S^n_j|} = \frac{|S^n_i \cap S^n_j|}{|S^n_i \cup S^n_j|}
\]

The algorithm computes the similarities between the paths using all possible orders and sums up all the results using weighing factor (the higher the order the bigger the weight) to give final similarity between the paths. This value is used as the fitness score in the genetic algorithm.

There can be many other ways of computing the fitness score given two paths. Another approach was proposed in [2]. In this paper, the fitness function is made
4.3. DATA SIMILARITY

of two parts. The first one is a measure of how many branches the two paths have in common from the start node until the point of deviation - $NC$. The second one indicates how much the input deviates in the point of deviation with the desired branch. The fitness score is then calculated as a ratio of the value of the predicate function associated to the branch - $EP$ (so for example if the branch had a condition $y == 100$ the value is $|y - 100|$) and the predicate function’s maximum value among the inputs that tried to reach that branch - $MEP$. Then the fitness function has a form:

$$F_t = NC - \left( \frac{EP}{MEP} \right)$$

More fitness functions are listed in [25]. There are also more meta-heuristic search algorithms such as tabu search, hill climbing, simulated annealing, or scatter search. However, they are better tailored for the goal-oriented test data generators.

4.3.2 Data similarity in execution reproduction

Path-oriented input generator algorithms are what we may use for finding similar data. For the purpose of the execution reproduction, we just need to skip the path selection step. A path we want to produce in an event is the one logged during the recording phase. Later, in the replay we execute the events with different sets of parameters, and the goal is to obtain a path that is not necessarily identical but close to the one from the recording phase. The procedure is repeated for every user event that we want to reproduce.

Definition

Logs produced after event execution with different parameters are indicators of whether data is similar. Because of this, they are used in our definition of data similarity.

Definition 14 (Data similarity definition). Let $rlog$ be a function that returns a reduced log after an event execution. A parameter $p_1$ of event $e_1$ that is executed in an initial state $S_0$ is similar to a parameter $p_2$ of the same event executed in an initial state $S'_0$ if $rlog(e_1(S_0, p_1)) = rlog(e_1(S'_0, p_2))$. We will denote it as $sim(e_1, (p_1, S_0), (p_2, S'_0))$.

The definition uses a reduced log that was defined in [3]. Persistence on the reproduction environment will typically contain less data, and that will cause a smaller number of iterations executed and logged. However, this does not influence the quality of our solution because we only remove the repetitive entries.

Similarity is defined not only over parameters but over a tuple of a parameter and an initial state. Logs represent execution paths which, as explained before, are affected by the parameters and initial states. Therefore, to obtain the same execution path, we need to configure them together. We could try to find similar data using fixed parameters and by simply changing initial states but that would be very hard to apply. In reproduction we have a fixed parameter and the initial state that was logged in the recording phase and we need to find such a parameter and an initial state that will cause a generation of the same log (after reductions of the logs).

Our strategy is to use a fixed initial state on the reproduction environment and to only try different parameters. Then our problem reduced to finding parameters that cause a certain execution path. This is what has been implemented in the path-oriented test input generator algorithms. As such we will use one of these algorithms to find similar data.
Data similarity and persistence

One thing we need to be aware of is that the described approaches do not mention how to deal with persistence when generating input data. The programs we use for our reproduction may use a persistent state and this can lead to complications with obtaining a specific path in a program. Let us consider the following example:

```csharp
bool UserExists(int Id)
{
    DataContext dx = new DataContext();
    var user = dx.GetUser(Id);
    if (user <> null)
        return true;
    else
        return false;
}
```

Originally the return value was `true`, however with the same parameter on the reproduction environment `false` is returned. This is due to the fact that the database in the reproduction environment does not have a user with a given `id`. In this case, an `id` must be found that exists in the database and causes the `if` statement to evaluate to `true`.

This complicates the way to automatically choose a proper input parameter. Without knowing the database state, there is a small probability of picking the right one. Using the last example, if the database contained only one user, the probability is \( \frac{1}{2^{32}} \) (where \( 2^{32} \) is the number of all possible integers). In testing, this is usually solved by using mock objects. We would create a database mock of `dx` that will return once null and the other times an instance of a user after calling `GetUser`. In execution reproduction we need to use a real database to be able to get the same error or a particular event execution that occurred in the original environment.

There are a number of papers that investigate the issue of persistence in test case generation. The solutions described in them are meant to work for a database as a persistence and a specific kind of a database - relational database.

The authors of use an extended symbolic execution in which a database state is treated as a special input parameter and is involved in a path condition calculation. A domain of each input parameter is defined by a path condition, and it is divided into sub-domains using boundary value analysis. Boundary values are extracted from a database schema and these can be: database types and lengths, unique, not null or default constraints, primary and foreign key constraints.

A tool called AGENDA populates a database using boundary value analysis and auxiliary files with sample inputs which are provided by a tester. The generated inputs are not meant to cover a specific path but rather a heuristics type chosen by a tester which can be duplicated, null or a boundary values heuristics.

In the approach is to use a concolic testing which runs the program on random inputs with some initial database state, as well as running it on symbolic inputs and a symbolic database. After executing the program with concrete inputs, a path’s constraint and a database’s constraint are constructed. The latter one contains information about the database meta data and the executed SQL queries. The inputs are calculated using a constraint solver and by updating the database state.

All these solutions work under certain conditions and still do not guarantee total path coverage. In this research we do not intend to find a solution to that problem. We assume the we have a set of inputs (can be manually provided by a tester) for the event test cases that are capable of achieving a high path coverage.
4.3. DATA SIMILARITY

Infeasible paths

The risk with path-oriented input generators is that there can be a path in a program that is infeasible and then it is impossible to find an input that invokes such a path. In case of execution reproduction, the path that was feasible in the original environment may be infeasible in the reproduction environment as was the case with the example in [4.3.2].

Identification of infeasible paths in a program is an undecidable problem [12]. Such an algorithm could be used to solve the halting problem. Existing solutions are partial, since they only work in special cases only. If we used symbolic execution, then path feasibility can be established by determining if the path condition is satisfiable (Constraint Logic Programming). If the constraints are contradictory, then the path is infeasible. A tool called PET is capable of automatically checking whether constraints are inherently contradictory [29]. This approach has some limitations and it is not capable of working with objects that are undefined at compile time, such as the user object in the example [4.3.2].

Paper [17] gives an empirically derived rule for when a path can be considered highly infeasible. If there are two conditional statements which are e-correlated then there exists at least one path that passes through them and is infeasible. The statements $x$ and $y$ are e-correlated when there is a path between them in a control flow graph and they do not have control dependency relationship (there exist a path between $x$ and $y$ such that except $x$ and $y$, $y$ postdominates each node from this path but $y$ does not postdominate $x$) and they are influenced by the same minimal set of variables.

If the algorithm for path selection uses heuristics, we can monitor if the outcomes of the iterations leads to an improvement. In the case of a genetic algorithm we can monitor the fitness function [1]. A lack of progress is identified when the following two criteria hold true: the increment in the best fitness for the consecutive $NL$ number of generations is less than some fixed value $\delta$; and the progress between the first and last generations of this sequence is lower than $\Delta$. $NL$ indicates how persistent the lack of progress should be to imply that the path is unfeasible. Its value should be higher for programs in which it is more difficult to generate proper inputs.

In our solution we assume the paths are feasible. As mentioned in the previous section, we are able to cover most of the paths thanks to the manually specified inputs.

Complex input generation

Events may take parameters that require a specific structure, for example objects representing a bit map, url, telephone number etc. Then it might be impossible to automatically generate such inputs without any manual help. There are testing tools such as T2 [20] that have a support for guidance on how the input data should be generated in such cases. In T2, a tester provides a so called custom domain generator for the type of a parameter. This is a class which inherits from the class of the investigated parameter, (so it remains all the methods the parameter has), and in the constructor it specifies how the objects should be properly created. After plugging the custom domain generator in T2, the input data can be automatically generated with valid instances of the special parameters.

T2's custom domain generators are used in the implementation of our solution.

Data similarity for event sequences

Finding similar data is the second phase of our event execution reproduction after removing commutative events. We have to find similar inputs for all remaining non-commutative events. Let the original sequence be
\[ S = [e_1(S_0, p_1), e_2(S_1, p_2) \ldots , e_n(S_{n-1}, p_n)] \]

We need to find a sequence of parameters \( P \) for the corresponding events in \( S \), where \( P = [p'_1, p'_2, \ldots , p'_n] \). If \( R = [e_1(S'_0, p'_1), e_2(S'_1, p'_2) \ldots , e_n(S'_{n-1}, p'_n)] \) is the reproduced sequence with the parameters from \( P \), then the following must hold true:

\[ \forall k : 1 \leq k \leq n, \exists S'_{k-1} \text{sim}(e_k, (p_k, S_{k-1}), (p'_k, S'_{k-1})) \]

In fact, to find the sequence \( P \) we also need to find the initial states in which each of the parameters is similar to the original ones. This will be a sequence \( PS = [(p'_1, S'_0), (p'_2, S'_1) \ldots , (p'_n, S'_{n-1})] \). The states are subsequent states in \( R \) and have a property that each next initial state is a result of the previous initial state. Therefore, we have to treat \( PS \) as one unit and we should not determine its elements in isolation. We have no control over the consecutive states in \( PS \) except the choice of the first state \( S'_0 \) which is the first initial state of the reproduced sequence \( R \).

To conclude, in the second phase of the reproduction it is necessary to find a proper initial state of the first executed event and a sequence of parameters that will be similar in the consecutive states with the parameters and initial states of the original sequence.

### 4.3.3 The algorithm

We use the genetic algorithm [4.3.1] for the implementation of data similarity. Given an input of a reduced original log with recorded events and their parameters as well as the execution paths, we try to find the sequence of parameters that will generate the same log in the reduced form on the reproduction environment. Here is how the algorithm works:

**Algorithm 2 Algorithm for finding similar data**

```plaintext
1: procedure FINDSIMILARDATA(Log originalLog)
2:   oEvents := getEvents(oLog)
3:   oParameters := getParameters(oLog)
4:   population := getInitialPopulation(oEvents, oParameters)
5:   for all \( g := 1 \rightarrow \text{maxGenerations} \) do
6:     survivors := getSurvivors(population, survivorsNumber)
7:     for all \( p := 1 \rightarrow \text{maxPopulationSize} \) do
8:       parents := getTwoRandomSurvivors(survivors)
9:       children := crossOver(parents, crossOverRate)
10:      children := mutate(children, mutationRate)
11:     population := population \cup children
12:   end for
13:   bestChromosome := getBestChromosome(population)
14:   if bestChromosome.fitness = bestFitnessScore then
15:     return bestChromosome
16:   end if
17: end for
18: return null
19: end procedure
```

This looks similar to any genetic algorithm. First, an initial population is created and then iteratively new generations are formed by the means of crossing over and mutation. This continues until one member of the population has the best fitness (in this case it is 10) or the evolved population has reached the maximum number of generations.
4.3. DATA SIMILARITY

The values of the constants used in the algorithm are in Figure 5.1. The constants crossOverRate and mutationRate direct how the search explores the solution space. The first one is a probability of making a cross over for a pair of chromosomes and should have a high value (around 0.7) because it causes the search to converge to what is currently the best solution. The second one is the probability of making a mutation on a chromosome, it should be very low because it causes the search to diverge from the best solution at the time. The maxPopulationSize determines the size of the search space and the value depends on the difficulty of finding the solution. In our case the difficulty is influenced by the number of events in a chromosome and the code complexity of the event implementation. The higher those values are, the higher the value of maxPopulationSize should be. The constant survivorsNumber is the number of surviving chromosome from the population. This value should be chosen depending on the population size. If it is a small fraction of the population size, then the population will be less diverse and only the best results will be used to produce the next generation. If we set the same value as the maxPopulationSize then we give a chance of evolution, even to the worst chromosomes and we make the algorithm run a greater number of times. The last constants: maxGenerations and bestFitnessScore define when the algorithm terminates. The former should have a high value, and the latter’s value depends on the implementation of the fitness function.

A chromosome is a sequence of parameters, a test case of the reproduced events. A population is a set of chromosomes, so a set of candidates for the reproduction test case. The most crucial point of the algorithm is a call to the function getSurvivors. In this function, based on the fitness score, we make a selection of the chromosomes that will be used to form a population of the next generation. This is also the point where the search is directed towards the best solution.

**Fitness function**

The fitness function measures how close a candidate solution is to the optimal result. In the case of data similarity, it is measured by comparing the generated reduced log with the original reduced log.

Fitness is calculated in line 12 of the algorithm when the call to getSurvivors is made. In that function we take the current population and for each chromosome which is a test case we set an initial state and execute the test case. This will cause the generation of a log which is then reduced. We compare this log with the reduced log of the original events. This is done by parsing the generated log and the original log to obtain the execution paths. The paths are compared according to definition. The closer they are, the lower the fitness score, which in this case means better fitness.

The total fitness of a chromosome could be a sum of fitness scores calculated for

**Figure 4.6: Algorithm constants**

<table>
<thead>
<tr>
<th>constant name</th>
<th>constant value</th>
</tr>
</thead>
<tbody>
<tr>
<td>crossOverRate</td>
<td>0.5</td>
</tr>
<tr>
<td>mutationRate</td>
<td>0.05</td>
</tr>
<tr>
<td>maxPopulationSize</td>
<td>50</td>
</tr>
<tr>
<td>survivorsNumber</td>
<td>10</td>
</tr>
<tr>
<td>maxGenerations</td>
<td>500</td>
</tr>
<tr>
<td>bestFitnessScore</td>
<td>10</td>
</tr>
</tbody>
</table>
each event in a test case that the chromosome represents. However, that can give some undesirable results. For example, if we have two chromosomes, represented as a sequence of parameters with the fitness score they give:

\[
[(p_1, 0), (p_2, 0), (p_3, 0), (p_4, 0), (p_5, 0), (p_6, 0), (p_7, 8)]
\]

\[
[(p'_1, 1), (p'_2, 1), (p'_3, 1), (p'_4, 1), (p'_5, 1), (p'_6, 1), (p'_7, 1)]
\]

If we sum up the fitness scores of the first sequence, we get a total fitness equal to 8 and for the second sequence it is 7. This would imply that the second chromosome is better and has a higher chance of survival to the next generation. Because the next step of population evolution is a cross over, which results in the replacement of part of the sequence with a sequence from another chromosome, the algorithm would merge faster towards the solution if the first sequence survived instead of the second one. If the cross over is done on the 7th position, then the fitness scores for the first 6 elements remain 0’s and the fitness of the 7th element can be improved.

For this reason, fitness of a chromosome is measured as a number of elements in a chromosome that have a fitness score 0. We use the sum of fitness scores as a secondary criteria in our algorithm. The first criteria is better with a higher value, and the opposite is true for the second criteria.

The function \( \text{getSurvivors} \) should not choose only the fittest but also a couple of less fit chromosomes. This is done in order to maintain diversity in a population and avoid running into a local maximum with the fitness function. Therefore we apply a roulette wheel selection for the choice of survivors. It is a selection in which the chance of being selected is proportional to the fitness but it does not guarantee the fittest members will go through to the next generation.

**Cross over**

Cross over is an operation which guides the search towards the optimal solution. We use a probability, defined in the algorithm as \( \text{crossOverRate} \), that indicates if the cross over should occur for the two given parents. If the cross over is skipped, then the children are the exact replicas of the parents. Otherwise, a position in the chromosome sequence is randomly selected and after dividing the two chromosomes at this position, the resulting subsequences are exchanged. For example, crossing over the two sample chromosomes at position 4 will result in two children chromosomes

\[
[(p_1, 0), (p_2, 0), (p_3, 0), (p_4', 0), (p_5', ?), (p_6', ?)]
\]

\[
[(p'_1, 1), (p'_2, 1), (p'_3, 1), (p'_4, 1), (p_5, ?), (p_6, ?), (p_7, ?)]
\]

The fitness score in the last elements of the children contains \(?\), because after cross over fitness needs to be recalculated.

**Mutation**

Mutation is an operation on a single chromosome and it is also made with a certain probability (typically low). In the algorithm it is defined with a \( \text{mutationRate} \). The way mutation works is the following we move through all the elements of a chromosome and change the value to a new one with the \( \text{mutationRate} \) probability. Mutation, just like the cross over, also makes sure the algorithm explores the search space, however, not always towards increasing fitness.
Chapter 5

Validation (experiment)

We implemented the described algorithms to validate our solutions. We used Java as the implementation language and the solution can only reproduce Java programs which have their source code available. Furthermore, the log injection is built as an Eclipse plugin, so it requires Eclipse IDE to be installed.

Event commutativity and data similarity were developed and tested independently. We have run them against a sample program (experiment subject). The implementation and results are described below.

5.1 Solution implementation

The solution can be divided into two main parts: the log injection project and the reproduction project. The first one instruments the source code of a sample program and also builds its control flow graph which is serialized and saved on the disk. The second one implements the algorithms for event commutativity and data similarity.

5.1.1 Log injection implementation

This can be done in two ways: by modifying a source code or by modifying a compiled code. On the example of Java, the modification of a compiled code would mean changing bytecode. There are libraries capable of doing that like [ASM] and [BECEL]. They are very powerful but using them can be difficult because of the bytecode syntax and direct operation on the stack. An abstraction allowing the use of Java to read and write bytecode - [javassist] is available, however, the possibilities are more limited and code injection can be done only on a method or a class level.

Aspect Oriented Programming has an elegant way of injecting a code. Using one of the Java extensions - [AspectJ], the developer only needs to specify special constructs that encapsulate a concern - aspects. Than the process called weaving automatically inserts the instructions to the bytecode. The problem is that AspectJ does not allow the definition of aspects over language constructs such as control flow statements.

The second way of injecting code is by a modification of a source code. The disadvantage it brings in comparison to the first method is that the changes are visible to a developer and may make the source code less readable. However, the implementation of code injection in this way is much simpler and easier to verify. For Java applications this can be done with the development of an Eclipse plug-in that uses [Java Development Tools]. JDT provides a powerful API that allows traversing the AST tree (including packages, projects, classes) and adding or modifying its nodes. The changed code can be saved to a separate file so that we do not face the
risk of breaking the original source code, and in case we need to repeat the code injection, we do not need to erase the injected statements manually.

Figure 5.1: Code injection using the bytecode in Java

Figure 5.2: Code injection using the source code in Java

We developed an Eclipse plugin that uses JDT and saves the instrumented code into a separate file. JDT allows walking of AST tree nodes with the use of a visitor pattern. We create an instance of ASTVisitor which has methods visit and endVisit implemented. These methods take an AST node as a parameter when the AST is traversed and a type of a visited node matches a type of method’s parameter, these methods are invoked. The method visit is executed when a node is reached and the endVisit is executed when all children of a node have been visited.

The logging code is injected in the implementation of a visit method. Additionally, when traversing an AST we also gather information needed for building a CFG. We create an intermediate data structure called block tree that represents the order and the nesting of a program’s statements. A block is a sequence of statements that belong to the same execution context. A tree structure is used to show how the blocks relate to each other. Here is an example of a block tree created for a method m:
A block tree is created during the traversal of an AST. We create a stack and when a node's method visit is executed, we create a block node representing the visited node and we put it on the stack and when endVisit is executed we pop the last block node from the stack and set its parent as the current top block node on the stack. Here is how the mechanism works on the previous example code:

Next, we replace the nodes that represent method calls with the corresponding bodies of the methods (so called inlining). From such a tree we can build a CFG. We traverse the block tree using DFS and we create a CFG node for each block node. If a node represents a control flow statement, we add two nodes: one for a control flow statement evaluated to true, and the other one for a control flow statement evaluated to false. The first node is named after the control flow statement and the second one has a "no" prefix added. On the example in Figure 6.5, for the if

Figure 5.3: A sample method and the corresponding block tree

Figure 5.4: The process of creating a block tree. Each rectangle represents a block node. When a block node is popped from the stack, then its parent is configured to be the top element on the stack.
statement we get a if CFG node and no if CFG node. An exception is an if - else control flow statement because it has an else defined so we do not need the no if because else already represents the alternative flow of the if. After creating a new CFG node, we set its parents as all the last nodes that were added to the CFG (the current leaves).

Figure 5.5: Transformation from a block tree graph into a CFG. On the left a block tree and on the right a CFG created from it. For simplicity we skip the inlining of the block tree in this example.

The "no nodes" are auxiliary and may be removed from the CFG. In the example above we can see the while does nto have an arrow that cycle back. This is because in our data similarity and event commutativity implementation we take into account only one loop iteration.

Some control flow statements like return, break, continue unconditionally change control flow and an execution context. They have to be refined in a graph by setting their child nodes properly (not to the node representing the next statement, but to the node representing the first node in the new context).

5.1.2 Event commutativity

In the implementations for both components we use T2 which is an automatic testing tool for Java. It works by generating random fixed - length sequences of methods from a given class. The inputs for the methods are generated randomly but there is a possibility to provide a set of predefined parameters by the use of custom object generators. A useful feature is a replay tool that allows recording test cases and replaying them later. We use T2 as an automatic test cases generator.

The event commutativity component takes as an input a list of user events and produces as an output a matrix of those events with values that indicate if a pair of events commute (boolean value true) or not (boolean value false), and a
5.1. SOLUTION IMPLEMENTATION

code coverage value that indicates how confident we can be with the result. The implementation is done in the following way:
Algorithm 3 Event commutativity implementation

1: **procedure** GENERATECOMMUTATIVITYMATRIX(UserEvent[] userEvents)
2:    matrix := initializeMatrix();
3:    eventPairs := getUniqueEventPairs(userEvents)
4:    for all eventPair from eventPairs do
5:        coverage := 0
6:        testLogs := {}
7:        mutantLogs := {}
8:        executionStartTime := now
9:        maxExecutionTime := maxTime(eventPair)
10:       while now - executionStartTime < maxExecutionTime do
11:           testCases := generateTestCases()
12:           for all testCase from testCases do
13:              resetState()
14:              mutatedTestCase := mutate(testCase)
15:              testLog := executeTest(testCase)
16:              resetState()
17:              mutantLog := executeTest(mutatedTestCase)
18:              testLogs := testLogs $\cup$ {testLog}
19:              mutantLogs := mutantLogs $\cup$ {mutantLog}
20:              newCoverage := getCoverage(testLog)
21:              if newCoverage $\neq$ coverage then
22:                  executionStartTime := now
23:              end if
24:              coverage := newCoverage
25:           end for
26:       end while
27:    commute := true
28:    for all testLog from testLogs do
29:       mutantLog := getMutantLog(testLog, mutantLogs)
30:       if testLog $\neq$ mutantLog then
31:          commute := false
32:          break
33:       end if
34:    end for
35:    matrix[eventPair] := (commute, coverage)
36: end for
37: return matrix
38: end procedure

First, we make 2-combinations with repetitions of all events from an input sequence. Then we have two separate phases: test logs generation; and test log analysis. In the first phase, test cases are generated until coverage does not improve within maximum execution time for an event pair.

In a method generateTestCase we use T2 to randomly generate sequences of events. The sequences are accepted if they contain consecutive events $e_1$ and $e_2$ in any order. We could direct T2 to generate only sequences $[e_1, e_2]$ or $[e_2, e_1]$ but then we would only test commutativity in one initial state. If other user events affect persistence, it is worth to also investigate if $e_1$ and $e_2$ commute after application state changes. Therefore, we set the length of generated sequences not to two but to some arbitrary number greater than two and let T2 randomly choose the events. This means the sequence may not always be accepted. The bigger the sequence length, the higher are the chances of acceptance but also the longer is the execution.
5.1. SOLUTION IMPLEMENTATION

Once a test case is found with an accepted sequence of events, we need to mutate it. This means the order of $e_1$ and $e_2$ has to be switched. Mutation can be done under a condition that if events have the same parameter types, a second event does not reuse the parameter created in the first event. Otherwise, some tests will need to be rejected.

The function `executeTest` makes use of the T2 replay tool to execute the found test and its mutant. It returns a log produced after the execution. The logs are stored in separate files which are named in such a way that it is easy to locate a test, a mutant, a test log and a mutant log.

The coverage of a test case is calculated in a method called `getCoverage`. We use path coverage because it guarantees that test cases have as many different execution paths as possible, and we use execution path in the definition of event commutativity. If the coverage does not change for some period of time, we finish the execution of a test case generation loop. The details of coverage calculation are described below.

In the log analysis phase a log of a test, and a log of a mutant are compared. If they are the same, we need to continue checking other logs, otherwise we found a test case that proved the events do not commute and we can set the result in the matrix as non-commutative.

**Coverage calculation**

For each commutativity result we provide a path coverage value. If the coverage value is low (below 60%), then we should consider checking commutativity once again because uncovered test cases may influence the commutativity result.

To calculate the coverage value we use a previously generated CFG from which we can obtain a list of possible paths for an event. Because loops in a program would cause an infinite number of possible paths, we only take into account one iteration. After each test execution we check which paths were covered in a generated log. If the investigated event pair is $e_1$ and $e_2$, we make the following calculation:

$$\text{coverage} = \frac{\#p\text{comb}}{\#p1 \times \#p2}$$

Here $\#p\text{comb}$ is the number of path combinations of $e_1$ and $e_2$ and $\#p1$ is the number of paths in $e_1$, and $\#p2$ is the number of paths in $e_2$.

This formula can become too strict for some event pairs. This happens especially when a control flow in an event depends not on input parameters, but on a persistent state. If additionally the event pair is commutative, we have a limited number of feasible paths. Let us consider an example of an event that gets orders from a database:

```csharp
event PrintOrders()
{
    DataContext dx = new DataContext();
    var orders = dx.GetOrders();
    foreach(var order: orders)
    {
        print(order);
    }
}
```
The number of possible paths is 2. If we test commutativity of \textit{PrintOrders} with itself, then according to the formula the coverage will be:

$$\frac{\#p_{\text{comb}}}{4}$$

We can never reach more than 50\% coverage in this case. If we have a test with the following event sequence \([\text{preEvents}, \text{PrintOrders}, \text{PrintOrders}, \text{postEvents}]\), where \text{preEvents} is a sequence of preceding events and \text{postEvents} is a sequence of succeeding events, the formula will always produce the same execution path in the first \text{PrintOrders} event and in the second \text{PrintOrders} event. This is because the events commute, hence the first event will always cover the same path as the second. The resulting coverage may seem to be not good enough but it is the maximum we can achieve.

The event \text{PrintOrders} has no parameters and the control flow depends only on the state of the orders table in the database. If \text{PrintOrders} had parameters and those parameters would determine control flow, then we could get more than 2 path combinations.

The number of path combinations can also be limited for non-commutative events when the first event determines the path of the second event, as is the case in the following event pair:

\begin{verbatim}
event InitializeOrders ()
{
  DataContext dx = new DataContext();
  var orders = dx.GetOrders();
  if (!orders-initialized())
  {
    orders.initialize();
  }
}

event AddOrder(var order)
{
  DataContext dx = new DataContext();
  var orders = dx.GetOrders();
  if (orders-initialized())
  {
    orders.add(order);
  }
}
\end{verbatim}

When a sequence \([\text{preEvents}, \text{InitializeOrders}, \text{addOrder}, \text{postEvents}]\) is executed, then the \textit{if} in the \text{addOrder} will always be evaluated to \textit{true} (unless an exception is thrown in \text{InitializeOrders} which we do not consider here). Therefore, we introduce a test sufficiency that is calculated in the following way:

$$\text{suff}(k) = \frac{\#p_{\text{comb}}}{\#p1 \times k}$$

Here \(k\) is the number calculated for each event pair and it represents the average number of feasible paths in event \(e_2\) after execution of event \(e_1\). Finding the exact value of \(k\) is equivalent to finding feasible paths which is an unsolvable problem. Therefore, \(k\) will only be an estimation of the number of feasible paths in \(e_2\). It needs to follow the following constraints:
5.1. SOLUTION IMPLEMENTATION

- k is never greater than #p2
- k is greater or equal to 1
- k increases with complexity of \( e_2 \)
- k decreases with an increasing number of control flow nodes that use the same persistence as was used in \( e_1 \)

A formula that reflects those constraints is the following:

\[
k = \begin{cases} 
#p2 - #n2common & \text{if } #n2common < #p2 \\
1 & \text{otherwise}
\end{cases}
\]

The \( #n2common \) is a number of nodes in a control flow graph of event \( e_2 \) that use the same persistence as in event \( e_1 \). By the word "use" we mean either read or write operation. It may happen that the value of \( #n2common \) is bigger than the number of all paths in event \( e_2 \). If this happens, we set the value of \( k \) to one. When the complexity of event \( e_2 \) grows, the number \( #p2 \) is also bigger and causes the value of \( k \) to increase as well. When the value of \( #n2common \) increases, the value of \( k \) decreases which fulfills the last formula constraint.

The purpose of \( k \) is to reduce the number of paths in event \( e_2 \) such that it represents only the number of feasible paths after execution of event \( e_1 \). The above formula for \( k \) will make this work well, if in a control flow graph of \( e_2 \) the proportion of control flow paths to control flow nodes is high. We could have the following control flow graphs of \( e_2 \):

![Diagram of control flow graphs](image)

**Figure 5.6:** Different kinds of control flow graphs of event \( e_2 \). The red nodes are examples of \( n2common \) nodes. They are identified manually by observing in the code the control flow nodes that depend on the persistence that is used in event \( e_1 \).

All these graphs are all built from 7 nodes but they have different number of paths. Graph a has one path, graph b has two paths and graph c has three paths. If we take two nodes as the \( n2common \) (marked as red), then we eliminate all paths in the case of a and b and in the case c we eliminate two paths and leave one possible path. Therefore, \( k \) gives the best estimation if there are many independent paths in a control flow graph. One subtracted node will correspond to one subtracted path.

We do not make a distinction between reading and writing to the same persistence when counting the \( n2common \) nodes. It may happen that event \( e_1 \) only does reading operations and then does not influence the number of paths in event \( e_2 \).
Moreover, even if \( e_1 \) was updating the persistent state that is later used in \( e_2 \), it may not always affect the path feasibility of \( e_2 \). In this way, we may eliminate the paths that were in fact feasible in \( e_2 \). Then, the number of covered paths \( \#pcomb \) can become higher than \( \#p1 \times k \). This will result in a total test sufficiency greater than 100%. This in fact happened during the experiment in the commutativity matrix generations and can be seen in the presented results.

We can expect some of the paths for a given event to be infeasible if control flow depends on a particular state of persistence. If we have the following event with A, B, C marking the control flow nodes:

```java
event PrintOrders()
(A){
(A) DataContext dx = new DataContext();
(A) var orders = dx.GetOrders();
(A) for(var order : orders)
(B) {
(B) order.print();
(B) }
(C)}
```

The path [A, C] which is feasible only if there are no orders in the persistence with orders. This is possible if for example we had a sequence of delete order events that would remove all orders and that sequence would be executed before \( \text{PrintOrders} \). We might not have such events and if we do it may be difficult to execute them in this particular order.

The way we treat this problem is by not forcing the coverage of these paths. Initially, we exclude them from the sufficiency calculation, however, if they turn out to be feasible we add them to the list of feasible paths and include them into the sufficiency calculation. In this way we do not lower the sufficiency by the paths that cannot be executed with a given persistent state.

Once a path that was not feasible is included into the sufficiency calculation, we need to cover all path combinations with that path. This will cause the value of sufficiency to fluctuate, going up and down as new feasible paths are discovered. That is why we execute an event pair until the sufficiency value does not change for a certain amount of time.

### 5.1.3 Data similarity

Data similarity component takes as an input a log that contains a sequence of original user events and their corresponding internal events. It returns a sequence of user events that were used in the input but with parameters adjusted to the reproduction environment. The solution is based on a genetic algorithm; we presented a prototype in 4.3.2.

A chromosome is a sequence of user events and their parameters that are a potential solution to the algorithm. Each time a chromosome is created, or an input of one of its events is changed, we calculate its similarity. The calculation is done by first resetting the persistence to an initial state and then executing the chromosome’s events. Even if only an input from one event is changed, the similarities of all events in the chromosome need to be re-calculated. This is because events in a chromosome are non-commutative meaning an execution of one of them affects the execution of the following events in a sequence. If a parameter is changed in one event, the similarity value for the consecutive events may be altered. If one of the events is re-executed, all the previous events in a chromosome need to be re-executed in order to always have the same initial state for the re-executed event. Therefore,
any change in a chromosome must be followed with a complete re-execution of all its containing events.

Each event execution will produce a log for which we first apply reduction and then compare it with the corresponding reduced original log. Based on that we set a similarity value for each event of a chromosome. The lower the similarity value, the more similar the logs will be. If the value is 0, this means the logs are the same. The similarity value for a chromosome is measured with a fitness function.

In a fitness function we use the number of events which do not have 0 similarity as the first criteria and total similarity as the second criteria. We use a fitness function in survivors selection which can be implemented in two ways. First, the survivors will be the parents of the new generation population. We want to use the best - fit chromosomes as parents but at the same time we want to maintain a diversity in a population by allowing the less - fit ones. Hence one way to get parents:

```
1: function GetSurvivors(population, populationSize)
2:   for all chr in population do
3:     chr.fitness := (getNonZeroSimilarity(chr) + getTotalSimilarity(chr)) × random
4:   end for
5:   population := orderByFitness(population)
6:   return Top(populationSize, population)
7: end function
```

Fitness is assigned to each chromosome as a sum of the number of non-zero similarity events (function getNonZeroSimilarity) and the total similarity (function getTotalSimilarity) which is a sum of similarities of all events in a chromosome. We use the non - zero similarity instead of the zero similarity to make the value compatible with the total similarity which is better when it decreases. The sum is multiplied by a random number which will make the value weighted. In this way a chromosome that is the best may not always be selected but has a high chance of being selected. The chance of selection is proportional to the fitness value but it does not exclude the less - fit chromosomes. As a consequence of this technique, the population becomes more diverse. The survivors are chosen by taking the top chromosomes after sorting the population.

A problem in this implementation of GetSurvivors might be that we sum up the two fitness criteria, therefore we treat them equally. This may cause problems we described in. The second way of selecting survivors it by "tournament selection".

```
1: function GetSurvivors(population, populationSize)
2:   while size(newPopulation) < populationSize do
3:     tournamentPopulation := pickSubsequenceRandomly(population, tournamentSize)
4:     tournamentPopulation := orderByFitness(tournamentPopulation)
6:   end while
7:   return newPopulation
8: end function
```

Here a set or chromosomes is treated as a sequence with each chromosome having a position index. From that list a subsequence of size tournamentSize is picked. The choice of tournamentSize determines the diversity in the population. The
smaller the value, the more diversity. The subsequence is chosen in the random position of the chromosomes sequence. It is ordered using fitness in the orderBy-Fitness method, however this time the two criteria are not summed but are given priorities instead. Having two chromosomes, if the first criteria - number of non zero similarity events is the same, then we apply the second criteria which is the total similarity value for the chromosome. This way the chromosomes with the least non zero similarities always have an advantage and are more likely to be selected.

In our implementation we tried both algorithms and they yielded similar results. However, we decided to use the "tournament selection" as it allows more flexibility on adjusting fitness calculation. This allowance comes from the fact in the first calculation we cannot use the similarity criteria prioritization. We need to first calculate a total similarity value for each chromosome so we can multiply it by a random value. In the second approach the randomization is done not with the similarity value but with the selection of the tournament population. We may then order chromosomes with or without criteria prioritization.

Running the experiment with such a fitness function still did not work very well. The algorithm could not find an optimal solution for sequences with more than 5 events and the population became monolithic after some generations. This was because chromosomes with most 0 similarity were too dominating. Let us consider the following small population of three chromosomes:

\[
\begin{align*}
&[(p_1, 0), (p_2, 0), (p_3, 0), (p_4, 0), (p_5, 0), (p_6, 3), (p_7, 5)] \\
&[(p_1, 0), (p_2, 0), (p_3, 0), (p_4, 0), (p_5, 0), (p_6, 4), (p_7, 9)] \\
&[(p_1, 1), (p_2, 1), (p_3, 1), (p_4, 2), (p_5, 1), (p_6, 0), (p_7, 0)]
\end{align*}
\]

The first two chromosomes will always be selected as parents and the third one has very small chances to survive and move on to the next generations. In this case it would be beneficial to take the first and the last chromosomes as survivors, because when we cross them over, their children have a higher chance of improving the end result. This is why we added a chromosome uniqueness to the fitness calculation. Chromosome uniqueness indicates how many of the 0 similarity events take the same position as any of the already added chromosomes to the new population.

A similarity calculation is based on the comparison of reduced logs (def. 3). These are the logs without repetitive entries which are caused by the loops or recursion. The reason why we apply the reduction is that we do not want to make the value of similarity worse if only the number of iterations is different between the original log and the reproduced one. This is likely to happen because reproduction persistence typically has less data than the original one.

Despite the log reduction, loops may still make the similarity worse than expected. This issue concerns the loops that have a more complex control flow in their body as in this example:

Multiple iterations of the for loop can produce an infinite number of possible sequences of log entires. In the given example the entries will depend on the state of the persistent orders. We could obtain the following log:

FOR.ID21
IFTHEN.ID11
FOR.ID21
IFTHEN.ID10
FOR.ID21
IFELSE.ID11

Here nothing can be reduced anymore because in each iteration the control flow is different. Now suppose that on the reproduced environment we get a log:

\[\text{similarity is better when similarity value is low}\]
5.1. SOLUTION IMPLEMENTATION

Take chromosomes $c_1$ and $c_2$

1. $getNZS(c_1) == getNZS(c_2)$
   - YES
   - NO
2. $getUniq(c_1) == getUniq(c_2)$
   - YES
   - NO
3. $getTS(c_1) <= getTS(c_2)$
   - YES
   - NO

Return $c_1$
Return $c_2$

**Figure 5.7:** Fitness comparison for two chromosomes $c_1$ and $c_2$. Function $getNZS$ calculates non-zero similarity. Function $getUniq$ calculates uniqueness. Function $getTS$ calculates total similarity.

Similarity based on the comparison of those logs will not be so good because we have 3 log entries out of 6 that are different. However, in fact the only difference between the logs is the order in which the control flow points are logged. Because each of the decision statements depends on the state of the persisting order, it is likely that those control flow points will be logged in a different order in the original log and the reproduced log. In the example the logs will be the same only if active orders and orders with no id had been written in the same order as persistent orders on both environments.

Due to the limitation of our log reduction procedure, we make a distinction between those logs, however, it would take less time to find similar data (or even be possible) if we made no distinction between them. One way to apply such a change would be to tweak log injection such that we log the end of the loop after each iteration so that we would be able to distinguish which log entries come from a loop iteration. Then we can treat those log entries as a set and not as a sequence.
foreach(order in orders)
{
    log("FOR", ID23);
    print(order);
}

Figure 5.8: On the left is a piece of program that logs "for" loop. On the right are the two possible logs generated from it and their reduced versions.

DataContext dx = new DataContext();
var orders = dx.getOrders();
for(var order in orders)
{
    log("FOR", ID21);
    if(order.id == null)
    {
        log("IFTHEN", ID10);
        print("unknown order id");
    }
    if(order.active)
    {
        log("IFTHEN", ID11);
        print("order active");
    }
    else
    {
        log("IFELSE", ID11);
        print("order inactive");
    }
}

and make a comparison. Another approach would be to map the log entries back to a CFG and check if the traversed paths for the two logs are the same. However, we leave this implementation as a future work and in our experiment we always take into account the order of the logged entries.
5.2 Experiment subject

We wrote a small console application to test the implemented solution. In the application we have a special class that contains user events which are public methods that can take parameters and return values. We created events with the following signatures:

- **public boolean addOrder(String username, String product, Integer amount);** - adds an order to the list of orders
- **public void deleteOrder(String username, String product);** - removes a row with the order
- **public double getTotalPrice(String username, String bonus);** - gets a total price for a user
- **public void addProduct(String newProduct, Double price, Integer inStock);** - adds a product to a list of products
- **public void updateProductName(String oldProduct, String newProduct);** - updates product names in all orders
- **public void updateProductInStock(String product, Integer newInStock);** - updates in stock amount for a product
- **public void deleteProduct(String product);** - deletes product if there are no orders
- **public void printProducts();** - iterates over all current products
- **public void cleanUpInvalidOrders(boolean cleanInactiveUsers, boolean cleanZeroAmount);** - removes orders that belong to inactive users and/or have amount 0
- **public String getUserStatistics(String username);** - gets the info about user orders

The events have different complexities. We use a number of paths and cyclomatic complexity value as a complexity indicator. The complexities for each of the events are listed in a table 5.9.

In the table we compare the number of paths with cyclomatic complexity which is sometimes different than the number of paths. It is because a cyclomatic complexity formula only counts the number of independent paths. The average cyclomatic complexity of all the events is 2.92.

In the last column there is a stable test sufficiency time. This is an estimation for the time until which test cases generation does not cause any coverage improvement. The value of stable test sufficiency time should be provided by a programmer of a reproduced program. To estimate it for an event we should take into account its execution time, complexity and the difficulty of selecting all paths given current persistent state and a set of possible input parameters.

In the event commutativity algorithm, having an event pair $e_1$ and $e_2$ with the corresponding stable test sufficiency times $t_1$ and $t_2$ we choose $max(t_1, t_2)$ for test cases generation.

The persistent state in our experiment consists of three persistence relations which are three csv files: users.csv, products.csv, orders.csv. Each user event trig-
<table>
<thead>
<tr>
<th>user event name</th>
<th>number of paths</th>
<th>cyclomatic complexity</th>
<th>stable test sufficiency time (in minutes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>deleteOrder</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>addProduct</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>updateProductInStock</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>printProducts</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>updateProductName</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>deleteProduct</td>
<td>3</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>addOrder</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>getTotalPrice</td>
<td>6</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>cleanUpInvalidOrders</td>
<td>7</td>
<td>6</td>
<td>10</td>
</tr>
<tr>
<td>getUserStatistics</td>
<td>10</td>
<td>5</td>
<td>12</td>
</tr>
</tbody>
</table>

Figure 5.9: Event complexities

This table was created for the sole purpose of the experiment validation. We will use it later when checking the results but it is not required in the execution reproduction process.

We can see that all events read from at least one of the files. 84% of the control flow points in all events depend on the persistent state. Therefore, we can expect that when one event writes to one of the files and the other event reads from the same file, the events have a high chance of being non-commutative.

5.3 Running the experiment and results evaluation

To run the experiment we need to first make some adjustments to the experiment subject program. We assume we have a source code of a program that we want to reproduce and we can load it with Eclipse. The first thing we need to do is to create a configuration file named log.properties which will contain paths to the
5.3. RUNNING THE EXPERIMENT AND RESULTS EVALUATION

original logs and paths to a directory where temporary files will be generated. The
detailed syntax of the configuration file is described in the manual.

Next, we need to instrument the code by running the log injector plugin. This
is done by selecting a class file that contains user events in the solution explorer.
Then when "Inject logs" is selected from the Logging Menu, the logs will be injected
to the user events and to the internal events belonging to the user events.

Figure 5.11: Log injection using the Eclipse plugin.

Each package in a project will have a copy created with a ".withlogging" added
to the name. It is a version of a package that contains all classes with the logging
injected.

In order to use custom domain generators, the types of the parameters in the
user events need to be replaced with reference types that extend a ReproParamter
class. For each of the newly created types we need to construct a generator which is
another class that is an extension of the type it generates. When all generators are
created, we need to create a so called interface map which is a class that contains
a mapping of parameter types to their generators. More guidelines can be found in
the T2 documentation.

Before each test case is executed, we need to make sure the program state is the
same every time. A developer of an input program needs to implement an abstract
class PersistenceResetter with its two methods: save, rollback. The first one is used
for saving current persistent state and the second one is used for bringing back the
persistent state to the one that was recently saved.

Event Commutativity

Once all the above steps are completed, we can test which events commute. This
can be done by writing a unit test in which we give a list of user events with sample
parameters. For each event we need to provide execution time (the execution times
for our experiment are given in a table 5.9).

We have 10 events and we need to test their 2-combinations with repetition. These are for example having events a, b, c pairs: (a, a), (a, b), (a, c), (b, b), (b, c), (c, c). The number of those event pairs combinations for 10 events is equal to \((10+2-1)\) which is equal to 55.

For each event combination T2 generates test cases until good test sufficiency
is reached. A test case is a sequence of 15 events and it needs to contain an event
pair occurring subsequently which is not easy to obtain because T2 generates event

\(^k\) combination with repetition is number of ways to sample \(k\) elements from a set of \(n\) elements allowing for duplicates.
sequences randomly. Some of the tests are rejected for the reasons described in 5.1.2. That is why the average test cases generation time for one pair is 13.5 minutes. This time does not include log analysis time. In total the execution of the whole experiment for event commutativity took over 16h on a 64-bit Windows machine with 2.54GHz processor and 4GB of RAM.

Figure 5.12 shows a resulting commutativity matrix for the first five events.

<table>
<thead>
<tr>
<th></th>
<th>add Order</th>
<th>delete Order</th>
<th>getTotal Price</th>
<th>add Product</th>
<th>update Product Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>add Order</td>
<td>false/4.2</td>
<td>false/1.8</td>
<td>false/3.1</td>
<td>false/1.8</td>
<td>false/3.0</td>
</tr>
<tr>
<td>delete Order</td>
<td>false/1.8</td>
<td>true/1.0</td>
<td>false/1.2</td>
<td>true/1.0</td>
<td>true/1.0</td>
</tr>
<tr>
<td>getTotal Price</td>
<td>false/3.1</td>
<td>false/1.2</td>
<td>true/2.3</td>
<td>false/1.5</td>
<td>false/1.4</td>
</tr>
<tr>
<td>add Product</td>
<td>false/1.8</td>
<td>true/1.0</td>
<td>false/1.5</td>
<td>false/2.0</td>
<td>false/2.0</td>
</tr>
<tr>
<td>update Product Name</td>
<td>false/3.0</td>
<td>true/1.0</td>
<td>false/1.4</td>
<td>false/2.0</td>
<td>false/3.0</td>
</tr>
<tr>
<td>update Product InStock</td>
<td>false/2</td>
<td>true/1.0</td>
<td>true/1.5</td>
<td>false/2.0</td>
<td>false/2.0</td>
</tr>
<tr>
<td>delete Product</td>
<td>false/2.8</td>
<td>false/1.2</td>
<td>true/2.3</td>
<td>false/2.0</td>
<td>false/3.0</td>
</tr>
<tr>
<td>print Products</td>
<td>true/1.0</td>
<td>true/1.0</td>
<td>true/0.9</td>
<td>false/1.0</td>
<td>true/1.0</td>
</tr>
<tr>
<td>cleanUp Invalid Orders</td>
<td>false/1.1</td>
<td>false/1.0</td>
<td>false/0.8</td>
<td>true/0.8</td>
<td>true/0.7</td>
</tr>
<tr>
<td>get User Statistics</td>
<td>false/2.3</td>
<td>false/1.3</td>
<td>true/1.3</td>
<td>true/0.9</td>
<td>true/0.8</td>
</tr>
</tbody>
</table>

**Figure 5.12:** Commutativity matrix (part 1). Value true means events commute and value false means events do not commute. After / sign is the test sufficiency value expressed as a fraction, so 0.5 means 50%. Red cells are are the results that are inconsistent relative to the perspective from the table 5.10 and will be discussed in more detail.

The event *addOrder* only commutes with event *printProducts* which is what we can expect because it writes to the orders persistent relation and reads from the products persistent relation.

The event *deleteOrder* commutes with events that do not use the orders persistent relation. An exception is event *deleteOrder* which we would expect not to commute with itself.

The event *getTotalPrice* only reads form persistence, therefore it should commute with events that only read from persistence as well, this includes events *printProducts*, *getTotalPrice* and *getUserStatistics*. However, we can see that events *UpdateProductInStock* and *deleteProduct* commute with *getTotalPrice* as well and those events also write to persistence.

The event *addProduct* commutes only with events that do not use products persistent relation. These are *deleteOrder*, *cleanUpInvalidOrders* and *getUserStatistics* according to the table 5.10 and these are the events with a true value for
addProduct event.

The event updateProductName has almost the same set of commutative events as addProduct because it only modifies products persistent relation. An exception is event printProducts which, based on the table 5.10, we would expect not to commute.

Figure 5.13 presents the second part of the commutativity matrix.

<table>
<thead>
<tr>
<th></th>
<th>update Product InStock</th>
<th>delete Product</th>
<th>print Products</th>
<th>cleanUp Invalid Orders</th>
<th>getUser Statistics</th>
</tr>
</thead>
<tbody>
<tr>
<td>add Order</td>
<td>false/2.0</td>
<td>false/2.0</td>
<td>true/1.0</td>
<td>false/1.1</td>
<td>false/0.8</td>
</tr>
<tr>
<td>delete</td>
<td>true/1.0</td>
<td>false/1.25</td>
<td>true/1.0</td>
<td>false/0.9</td>
<td>false/1.3</td>
</tr>
<tr>
<td>get Total Price</td>
<td>true/1.5</td>
<td>true/2.3</td>
<td>true/0.9</td>
<td>false/0.8</td>
<td>true/1.3</td>
</tr>
<tr>
<td>add Product</td>
<td>false/2.0</td>
<td>false/2.0</td>
<td>false/1.0</td>
<td>true/0.8</td>
<td>true/0.9</td>
</tr>
<tr>
<td>update Product Name</td>
<td>false/2.0</td>
<td>false/3.0</td>
<td>true/1.0</td>
<td>true/0.7</td>
<td>true/0.8</td>
</tr>
<tr>
<td>update Product InStock</td>
<td>true/2.0</td>
<td>false/2.0</td>
<td>true/1.0</td>
<td>true/1.0</td>
<td>true/0.9</td>
</tr>
<tr>
<td>delete Product</td>
<td>false/2.0</td>
<td>false/3.0</td>
<td>false/1.0</td>
<td>false/0.9</td>
<td>true/1.2</td>
</tr>
<tr>
<td>print Products</td>
<td>true/1.0</td>
<td>false/1.0</td>
<td>true/1.0</td>
<td>true/0.8</td>
<td>true/1.0</td>
</tr>
<tr>
<td>cleanUp Invalid Orders</td>
<td>true/1.0</td>
<td>false/0.9</td>
<td>true/0.8</td>
<td>false/0.6</td>
<td>false/0.9</td>
</tr>
<tr>
<td>getUser Statistics</td>
<td>true/0.9</td>
<td>true/1.2</td>
<td>true/1.0</td>
<td>false/0.9</td>
<td>true/1.8</td>
</tr>
</tbody>
</table>

Figure 5.13: Commutativity matrix (part 2). Value true means events commute and value false means events do not commute. After / sign is the test sufficiency value expressed as a fraction, so 0.5 means 50%. Red cells are the results that are inconsistent relative to the perspective from the table 5.10 and will be discussed in more detail.

The event updateProductInStock should have the same results as event updateProductName because it only writes to products persistent relation. We can see in red two events: getTotalPrice and updateProductInStock. They turn out to be commutative but should not be commutative because they also rely on products persistent relation. Just as for updateProductName, event printProducts is listed as commutative which might not be correct.

The event deleteProduct writes and reads from products persistent relation and only reads from orders persistent relation. This means it should only commute with events that only read from orders persistent relation and do not read or write to products persistent relation. There is only one such event - getUserStatistics which we can see is commutative. The rest of the event should be not commutative, this is exactly what we observe, except for the event getTotalPrice.

The event printProducts only reads from products persistent relation, so it commutes with all events except the ones writing to products persistent relation. This
happens except the event \textit{updateProductName} and \textit{updateProductInStock} which we described before.

The event \textit{cleanUpInvalidOrders} reads and writes to the orders persistent relation. There are four events that do not use the orders persistent relation: \textit{addProduct}, \textit{updateProductInStock}, \textit{updateProductName}, \textit{printProducts}. These are the only commutative events with \textit{cleanUpInvalidOrders} according to the matrix.

The event \textit{getUserStatistics} only reads from the orders persistent relation, hence it should not commute with events that write to the orders persistent relation which are: \textit{deleteOrder}, \textit{addOrder} and \textit{cleanUpInvalidOrders}. As we can see in the matrix these are the only events that do not commute with event \textit{getUserStatistics}.

The test sufficiency value in the commutativity matrices varies between 0.6 and 4.2. Values above 1 mean that we covered more than expected. This may happen because we use an estimation in the calculation of the test sufficiency value. It may turn out that the paths we assumed that were infeasible, were in fact feasible.

We marked with red the results in the matrix that may bring some doubt about the correctness of the result that we would expect after looking at the table. These are the pairs:

1. \textit{printProducts} - \textit{updateProductName} - true/1.0
2. \textit{printProducts} - \textit{updateProductInStock} - true/1.0
3. \textit{getTotalPrice} - \textit{updateProductInStock} - true/1.5
4. \textit{deleteOrder} - \textit{deleteOrder} - true/1.0
5. \textit{getTotalPrice} - \textit{deleteProduct} - true/2.3

All these pairs commute with high test sufficiency (min 100%) and they depend on the same persistent relation, thus they may influence each other. For the case of event pairs 1, 2 and 3, either \textit{updateProductName} or \textit{updateProductInStock} may influence \textit{printProducts}. However, the influenced aspect is on the result (how the event is printed on a console) and not on the control flow. As such, in accordance with our definition of event commutativity, the commutativity result for those pairs is in fact correct.

The event \textit{deleteOrder} deletes an order from the persistent orders relation. There are two control flows possible. If there is an order with a given user name and product name, the order gets deleted. Otherwise, the event does nothing. We could have the following test case that would show that the events do not commute: \{\textit{deleteOrder}("joanna", "lamp"), \textit{deleteOrder}("joanna", "lamp")\}. The first event deletes the order, however, the second does not. When we reverse the order of execution, the second event which becomes the first deletes the order. The control flow of the second event has changed in the reversed test case. According to the event commutativity definition, this proves the events do not commute. However, in the matrix we can see the events commute with high test sufficiency.

Even though we got a high test sufficiency, we did not cover a test case that reveals non-commutativity as the one above. This is because covering all possible paths may not be sufficient to discover such a test case. Let us consider a test case that has the same control flow as the previous one. Such a test case is built using the following event sequence \{\textit{deleteOrder}("joanna", "lamp"), \textit{deleteOrder}("monika", "table")\} and is executed with the following initial state of the persistent orders relation:

<table>
<thead>
<tr>
<th>username</th>
<th>product</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>joanna</td>
<td>lamp</td>
<td>1</td>
</tr>
</tbody>
</table>
The first event deletes an order but the second does not because there is no order for user "monika" and product "table", just as in the previous test case. When we reverse the order of the execution, the first event that was the second one - deleteOrder("monika", "table") cannot delete the order just as in the non-reversed sequence. The second event that was the first one - deleteOrder("joanna", "lamp") can still delete the order. Therefore, despite covering the same path as the previous test case, this time we did not prove the event deleteOrder does not commute with itself.

Event pair 5 follows the same reasoning. We have two control flows depending on the state of the products persistence relation in the event getTotalPrice: a product from a given order exists or does not exist. The second event deleteProduct can influence the control flow of the getTotalPrice by deleting a product as in the following test case [deleteProduct("lamp"), getTotalPrice("joanna")]. If we had only one order like in the products persistent relation above, then the control flow in the getTotalPrice would be different after reversing the order of those events. However, if we had a test case [deleteProduct("lamp"), getTotalPrice("monika")], executed with the same initial state, we would cover the same path as in the previous example but the reverse order test case would not cause any changes in the control flow of these events.

The results of event pairs 4 and 5 shows the limitation of our approach. Control flow that we use in the definition of event commutativity is just an abstraction of the real effects of an execution. It cannot, for example, fully capture the effect on the persistent state. For that reason our event commutativity algorithm uses a time limitation instead of reaching the maximum coverage. This still may give incorrect results such as the event pair 4 and 5. To make sure we generated enough test cases, we would need to include a persistent state as the context in which the events commute. This would however, create the problem of defining a metric over a persistent state. We leave this issue for further investigation and in the current implementation we need to set such a test case generation time that we can obtain at least high path coverage and cover as many different states of persistence as possible.

Data similarity

We implemented four test cases in order to test the data similarity algorithm. Each test case contains a sequence of event calls from the experiment subject program and represents some scenario that could have happened in the original environment.

First, we need to generate "original logs" which will contain the control flow that we later try to reproduce. This requires the creation of the "original persistence" which in our case will be creating new orders, users and product persistent relations and filling it with sample data that is necessary to run the scenarios. Once this is done, we can execute events. This will cause the generation of logs to the location pointed in the configuration file.

Next, we can run the algorithm that will now use different persistence. The execution runs until similar data is found or the maximum generations are reached.

In the reproduction we use the following persistence:
We have active and inactive users in the persistent users relation. Some of the users have no orders and some have more than one order; there are also two test users. In the persistent products relation we have products with no items in stock and products with 200 items in stock. Product "les5" was created as a representative of a product with a name that has less than 5 characters. The persistent orders relation contains orders of existing, not existing and test users and orders with 0 amount. The choice of such data was to enable all the possible control flows in all reproduced events.

The first scenario is: the last event tries to update a product that does not exist. A test case for that scenario consists of the following event sequence:

```plaintext
addOrder("joanna", "pencil", 5)
deleteProduct("ballpoint")
updateProductName("ballpoint", "Pen")
```

The events are executed with the original persistence as:

```
<table>
<thead>
<tr>
<th>name</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>john</td>
<td>true</td>
</tr>
<tr>
<td>sally</td>
<td>true</td>
</tr>
</tbody>
</table>
```

As we can see, the original persistence does not have a lot of data. However, the data here is sufficient for the purpose of this scenario. The three events use
only this data and if we had more records, this would not affect the execution of the events.

An existing user adds an order to an existing product that is in stock. Then an existing product that has no orders is deleted and after that someone tries to update the name of that product. In the reproduction environment we obtained the following test cases with similar data:

```java
addOrder("sally", "noorder", 10)
deleteProduct("les5")
updateProductName("les5", "les5")
```

```java
addOrder("inactive", "noorder", 6)
deleteProduct("les5")
updateProductName("ballpoint", "Pen")
```

Both of these test cases managed to trigger the same control flow as the original test case. There can be many more possible test cases. The first test case resembles the original one in a way that we try to update a product name that was deleted in the previous event. In the second test case the parameters of the event `updateProductName` have not changed, despite there is no "ballpoint" product in the persisting products for the reproduction. Still we managed to get the same control flow because the "ballpoint" that we try to delete is not in the persisting products. The difference between those test cases is that in the second one we do not see the interaction between the `deleteProduct` and `updateProductName`. Though the second test case gives the same similarity value as the first test case, it may provide less understanding to a developer making the execution reproduction. It is not demonstrated in the second test case that we update a product that was deleted in the previous event. We would need to additionally attach information about how the persistence changes with each event execution which is something that would require a separate investigation.

The second scenario involves a deletion of a product that has some orders. The original test case is the following:

```java
addProduct("pencil", 23.9, 12)
updateProductInStock("pencil", 7)
updateProductName("ballpoint", "Pen")
addOrder("monika", "Pen", 1)
deleteProduct("Pen")
```

The events are executed with the original persistence as:

<table>
<thead>
<tr>
<th>name</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>monika</td>
<td>true</td>
</tr>
</tbody>
</table>

Table 5.7: Users persistent relation.

<table>
<thead>
<tr>
<th>name</th>
<th>price</th>
<th>in stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>ballpoint</td>
<td>25.5</td>
<td>20</td>
</tr>
<tr>
<td>pencil</td>
<td>10</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.8: Products persistent relation.

The first event tries to add a product with a name that already exists in the persisting products. In the following event the "in stock" value is updated for that
product. In the next event a name is updated for one of the existing products. Then, an order is made with an active user for the product with the updated name. This product is then attempted to be deleted but because in the previous event an order had been made using it, the deletion will not be successful. We obtained the following test cases that have the best similarity:

```
addProduct("les5", 60.5, 30)
updateProductInStock("shoes", 30)
updateProductName("noorder", "nonexisting")
addOrder("inactive", "nonexisting", 3)
deleteProduct("pijama")
```

```
addProduct("shoes", 3.6, 6)
updateProductInStock("shoes", 6)
updateProductName("les5", "shoes")
addOrder("noorders", "noorder", 6)
deleteProduct("nonexisting")
```

Both of these test cases yield the same control flow as the original test case. When comparing the above test cases, the parameters are very different. The final event `deleteProduct` tries to delete "pijama" or "nonexisting" which have orders in the original orders persistent relation. Just as with the original test case, here we try to delete a product that has some orders. However, unlike in the original test case, these orders have been inserted before the test case execution. Therefore, just as in the solution for the previous scenario, we do not see how the previous events affect the last event - `deleteOrder`. We removed all the orders from orders persistent relation to force generation of such an input in event `addOrder` that will also be used in the event `deleteProduct`. In the first generation we managed to obtain a test case in the same manner as in the test cases above.

The third test case scenario involves getting user statistics when no orders for a user are given. The original test case is the following:

```
addProduct("desk", 214.9, 5)
addOrder("joanna", "desk", 1)
addOrder("joanna", "lamp", 0)
addOrder("joanna", "mouse", 5)
deleteOrder("joanna", "desk")
cleanUpInvalidOrders(true, true)
getUserStatistics("joanna")
```

The events are executed with the original persistence as:
5.3. **RUNNING THE EXPERIMENT AND RESULTS EVALUATION**

<table>
<thead>
<tr>
<th>name</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>joanna</td>
<td>true</td>
</tr>
<tr>
<td>sergio</td>
<td>true</td>
</tr>
<tr>
<td>george</td>
<td>false</td>
</tr>
</tbody>
</table>

Table 5.10: **Users persistent relation.**

<table>
<thead>
<tr>
<th>name</th>
<th>price</th>
<th>in stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>pencil</td>
<td>25.5</td>
<td>20</td>
</tr>
<tr>
<td>pen</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>book</td>
<td>5.99</td>
<td>5</td>
</tr>
<tr>
<td>mouse</td>
<td>2.30</td>
<td>4</td>
</tr>
<tr>
<td>lamp</td>
<td>13.8</td>
<td>0</td>
</tr>
<tr>
<td>agenda</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>paint</td>
<td>2.4</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5.11: **Products persistent relation.**

<table>
<thead>
<tr>
<th>user name</th>
<th>product name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>george</td>
<td>shoes</td>
<td>1</td>
</tr>
<tr>
<td>sergio</td>
<td>lamp</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 5.12: **Orders persistent relation.**

A new product is added and a user orders the new product. The same user makes an order but with a 0 amount which is not valid. Then, this user makes another order but with an amount greater than is in stock which is also invalid. Next, this user deletes the first order with the new product. The invalid orders are cleaned from the persistent orders relation. In the end, the orders information (statistics) is gathered for the user that was making all the described orders. The only order that had been successfully added was later deleted, therefore, the last event will return no results. These are the test cases that yield the same control flow by using persistence in the reproduction environment:

```java
addProduct("nonexistingProduct1", 0.4 , 6)
addOrder("noorders2", "nonexistingProduct1", 0)
addOrder("inactive", "nonexistingProduct1", 201)
addOrder("noorders2", "dress", 201)
deleteOrder("nonexisting", "shoes")
cleanUpInvalidOrders(true, false)
getUserStatistics("noorders")
```

```java
addProduct("nonexistingProduct2", 3.6, 30)
addOrder("testactive", "les5", 6)
addOrder("john", "shoes", 88)
addOrder("noorders2", "les5", 201)
deleteOrder("nonexisting", "shoes")
cleanUpInvalidOrders(true, false)
getUserStatistics("noorders")
```

In both test cases we managed to obtain the same control flow as in the original test case. Despite the two events, `cleanUpInvalidOrders` and `getUserStatistics`, which contain loops with control flow statements in the loop bodies which may cause execution reproduction problems as described in [5.1.2](#). The semantics of the loop in the `getUserStatistics` event is iterating over the orders and checking if an order belongs to a specified user. Because the user has no orders, the loop body is not executed at all, so we managed to find an input that causes the same control
flow for that event. The event `cleanUpInvalidOrders` has a loop body with the control flow that can be manipulated with the input parameters. This is why we also managed to reproduce the control flow for this event.

The last scenario we implemented tries to add an order when there are not enough products in stock. The original test case is the following:

```java
addProduct("desk", 214.9, 5)
addOrder("joanna", "desk", 1)
addOrder("sergio", "lamp", 1)
deleteOrder("joanna", "desk")
addOrder("joanna", "lamp", 2)
updateProductInStock("lamp", 2)
deleteProduct("tv")
cleanUpInvalidOrders(true, true)
getTotalPrice("joanna", "25%")
addOrder("joanna", "lamp")
```

The events are executed with the original persistence as:

```
<table>
<thead>
<tr>
<th>name</th>
<th>active</th>
</tr>
</thead>
<tbody>
<tr>
<td>joanna</td>
<td>true</td>
</tr>
<tr>
<td>sergio</td>
<td>true</td>
</tr>
<tr>
<td>george</td>
<td>false</td>
</tr>
</tbody>
</table>
```

Table 5.13: Users persistent relation.

```
<table>
<thead>
<tr>
<th>name</th>
<th>price</th>
<th>in stock</th>
</tr>
</thead>
<tbody>
<tr>
<td>pencil</td>
<td>25.5</td>
<td>20</td>
</tr>
<tr>
<td>pen</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>book</td>
<td>5.99</td>
<td>5</td>
</tr>
<tr>
<td>mouse</td>
<td>2.30</td>
<td>4</td>
</tr>
<tr>
<td>lamp</td>
<td>13.8</td>
<td>3</td>
</tr>
<tr>
<td>agenda</td>
<td>9</td>
<td>1</td>
</tr>
<tr>
<td>paint</td>
<td>2.4</td>
<td>10</td>
</tr>
</tbody>
</table>
```

Table 5.14: Products persistent relation.

A new product is added and a user makes an order for that product. Then, another user orders a different existing product. In the next event the previous user deletes the order that has been made in the second event and then makes an order for the same product as the user from the second order. Then, the "in stock" value is increased for that product. The event `deleteProduct` attempts to delete a product that does not exist and in the next event the orders persistent relation are cleaned from invalid orders. Next, the first user queries the total price for the orders and in the last event still adds an order but the product is out of stock.

```java
addProduct("nonexistingProduct1", 4.6, 6)
addOrder("testinactive", "dress", 0)
addOrder("sally", "shoes", 6)
deleteOrder("nonexisting", "shoes")
addOrder("john", "dress", 3)
updateProductInStock("dress", 1)
```
deleteProduct("nonexistingProduct2")
cleanUpInvalidOrders(true, true)
getTotalPrice("noorders", "")
addOrder("noorders2", "shoes", 201)

addProduct("nonexistingProduct1", 4.6, 1)
addOrder("john", "les5", 0)
addOrder("inactive", "shoes", 3)
deleteOrder("nonexisting", "shoes")
addOrder("john", "socks", 6)
updateProductInStock("les5", 6)
deleteProduct("nonexistingProduct2")
cleanUpInvalidOrders(false, true)
getTotalPrice("testinactive", "")
addOrder("testinactive", "dress", 88)

In the second test case the we did not manage to reproduce control flow for the event cleanUpInvalidOrders. This happened in two out of six runs of the experiment. The same control flow as the original for that event could only be achieved if the parameters were only true and true, and none of the previous addOrder events added an order with 0 amount, or for an inactive user. Obtaining these parameter with such a prior events can be difficult to achieve. The initial population is generated randomly, hence, if by a stroke of luck we get many true–true combinations, then we would have greater chances of finding an optimal solution. Otherwise, only a mutation could change the input, but this happens with a low probability. Because of this the result for this scenario was not always optimal.

We also ran the experiment with test case of length 15, 20, 30, 40 and 50 in order to check its performance. The results with execution times are in the table 5.14. Initially, we had a problem with reproducing a control flow for a test case of length 7 due to the fitness criteria being defined in the wrong way. At the beginning the primary criteria was uniqueness and the secondary was the non-zero similarity. Swapping this order significantly improved the results. This shows the importance of choosing appropriate fitness measures.

Each event has its parameters created by the input generators that pick the parameters randomly from a pre-defined set of parameters. If the probability of selecting the parameters generating a particular control flow is low, we need to generate more test cases to be able to cover this control flow in one of the test cases. This is also necessary if the event’s control flow depends on more than one parameter. This should be reflected in the value of a population size. This value will determine the size of an initial population. Because cross over is the main search mechanism in a genetic algorithm and mutations happen with a very low probability, it is important to have a good representative set of test cases in the initial population in order to find an optimal solution.

For all four scenarios we used 50 as the population size which seems reasonable, given the number of possible inputs and the number of control flow paths depending on the input parameters. Here is an overview of the number of possible inputs generated by the input generators:

Based on that we can calculate how many test cases can be generated using all of those possible inputs for a scenario. Let us take the last scenario as an example:

The total number of all possible test cases using all these inputs is equal $32 \times 10^{19}$ with respect to the fourth scenario. There is more than one solution among these
test cases. For example, in the first event \textit{addProduct} only one parameter affects the control flow, so instead of generating 125 test cases for that event we could simply provide 5 and find a solution. This would however require a manual input or implementation of a tool that could detect if a parameter is used in any of the control flow statements.

We ran the experiment for each scenario at least three times to make sure the results are not due to a "lucky" initial population generation. Here is an overview of the performance of generating and analyzing the test cases for all the scenarios we described:

### Table 5.16: Parameters and number of possible inputs generated by the input generators.

<table>
<thead>
<tr>
<th>parameter</th>
<th>possible inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>username</td>
<td>8</td>
</tr>
<tr>
<td>productname</td>
<td>5</td>
</tr>
<tr>
<td>price</td>
<td>5</td>
</tr>
<tr>
<td>in stock</td>
<td>5</td>
</tr>
<tr>
<td>bonus</td>
<td>5</td>
</tr>
<tr>
<td>amount</td>
<td>5</td>
</tr>
<tr>
<td>true or false</td>
<td>2</td>
</tr>
</tbody>
</table>

### Table 5.17: Parameters and number of possible inputs generated by the input generators.

<table>
<thead>
<tr>
<th>user event name</th>
<th>possible inputs</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>addProduct</td>
<td>$5 \times 5 \times 5$</td>
<td>125</td>
</tr>
<tr>
<td>addOrder</td>
<td>$8 \times 5 \times 5$</td>
<td>200</td>
</tr>
<tr>
<td>addOrder</td>
<td>$8 \times 5 \times 5$</td>
<td>200</td>
</tr>
<tr>
<td>deleteOrder</td>
<td>$8 \times 5$</td>
<td>40</td>
</tr>
<tr>
<td>addOrder</td>
<td>$8 \times 5 \times 5$</td>
<td>200</td>
</tr>
<tr>
<td>updateProductInStock</td>
<td>$5 \times 5$</td>
<td>25</td>
</tr>
<tr>
<td>deleteProduct</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>cleanUpInvalidOrders</td>
<td>$2 \times 2$</td>
<td>4</td>
</tr>
<tr>
<td>getTotalPrice</td>
<td>$8 \times 5$</td>
<td>40</td>
</tr>
<tr>
<td>addOrder</td>
<td>$8 \times 5 \times 5$</td>
<td>200</td>
</tr>
</tbody>
</table>
5.4 Conclusion

We implemented and tested event commutativity and data similarity components separately. They are also meant to run separately because it takes many hours to generate the commutativity matrix and it is only necessary to do it once. Having this matrix, it is easy to see which events can be eliminated from the execution reproduction process. For the remaining events a test case is generated with similar input data. Running this test case is the actual execution reproduction.

Event commutativity and data similarity components rely on control flow. It is information that is relatively easy to record and replay, even on a different environment. Logged control flow is able to give an insight about which parts of the code was executed and in which order. Reproducing control flow can be helpful in debugging programs that have more than one path. The quality of the execution reproduction grows with the complexity of the reproduced program. However, with more complexity also comes a greater degree of manual effort (pre-populating persistence, providing inputs for input generators) that needs to be put before the execution reproduction and a longer execution time of the two components.

The control flow did not always yield the expected results in the investigation of event commutativity as we saw in the resulting matrix of the experiment. This is because events commute depending on the execution context which can be the state of persistence. In our solution we define event commutativity regardless the context. When the event sequence is reduced by removing commutative events we may remove events that in the execution reproduction context do not commute.

Using control flow we were able to discover commutativity between events that share the same persistence but their operations do not affect each other. This is something that we would be not have been able to distinguish using static analysis.

The definition of data similarity is based on a control flow as well. As we have observed in the experiment, obtaining the same control flow with different
persistence may be difficult if we iterate over the records in a loop that has control flow statements in a loop body. Regarding this, our definition of data similarity may be too strict for some kinds of programs.

There can usually be more than one similar data possible as an event input, hence we can obtain multiple test cases for execution reproduction with the best similarity value. Some of these test cases have the advantage that they show the interactions between events that also occurred in the original execution. Since we cannot express those interactions with control flow, it is not indicated with the data similarity value that such test cases could be better candidates for an execution reproduction. To do that, we would need to log the changes made to persistence after event execution. Logging persistence is however something we try to avoid in our approach.

5.4.1 Future work
Several questions arose during the research and were not covered in this investigation, these are added to the future work.

1. How can we include persistence context to the event commutativity definition?

2. Can the quality of data similarity be affected if the order of entries logged in a loop body is not preserved?

3. How can we give priority to inputs that show interaction between events in selection of similar data?

4. How can we apply the event commutativity and data similarity to programs written in declarative languages?

The first question came after analysis of the incorrect results on event commutativity for some event pairs. As we saw, the same control flow can give different commutativity results and it is because of the state of persistence in which the events are tested. Incorporating persistence to the event commutativity definition would make the results more reliable.

The second point was brought up after a difficulty with obtaining similar data for event cleanUpInvalidOrders. The reason was that this event had a complex control flow in the loop body and it is difficult to reproduce such a control flow. We can make execution reproduction possible for such events if we disregard the order of logged entries for this control flow. It is not certain if this will affect the quality of the reproduced test cases.

The third point is also related to finding similar data and the answer to that question would improve the quality of the reproduction. A developer performing it would not only be able to reproduce the same control flow but also see how the previous events affected the control flow of the following events.

As far as the last point is concerned, we tested our solutions only on Java programs. For programs written in other OO languages the solution should be easy to adjust. It would be worth investigating if we can do that as well for programs written in declarative languages where a control flow is obscured in a program.
Bibliography


