SAARLAND UNIVERSITY

Faculty of Mathematics and Computer Science
Department of Computer Science
BACHELOR THESIS

Runtime Verification for Algorithmic Fairness

submitted by
Tobias Wagenpfeil
Saarbrücken
April 2023

Advisor:
Julian Lukas Siber, Jan Eric Baumeister
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Reviewer 1: Prof. Bernd Finkbeiner, Ph.D.
Reviewer 2: Dr. Rayna Dimitrova

Saarland University
Faculty MI – Mathematics and Computer Science
Department of Computer Science
Campus - Building E1.1
66123 Saarbrücken
Germany

Eidesstattliche Erklärung

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

Statement in Lieu of an Oath

I hereby confirm that I have written this thesis on my own and that I have not used any other media or materials than the ones referred to in this thesis.

Saarbrücken, April 28, 2023,

(Tobias Wagenpfeil)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

I agree to make both versions of my thesis (with a passing grade) accessible to the public by having them added to the library of the Computer Science Department.

Saarbrücken, April 28, 2023,

(Tobias Wagenpfeil)

Acknowledgements

Before I start addressing the topic of this thesis, I would like to thank Prof. Bernd Finkbeiner for making this thesis possible.
Further, I want to express my thankfulness to my advisors Julian Siber and Jan Baumeister not only for their countless professional advice but also for their moral support and the push to always go further.
Furthermore, I want to thank Prof. Bernd Finkbeiner and Dr. Rayna Dimitrova for reviewing my thesis.
Many thanks go to Michael Kormann and Stefan Wagenpfeil for proofreading this thesis.
Further, I want to thank my family and friends for their endless support, especially for enduring me with all my moods.
Last but not least I want to give special thanks to Tristan Didion, Ulysse Planta, and Niklas Klingler for the coaching sessions in the evenings.

Abstract

Throughout our lives, we are increasingly subjected to decisions made by algorithms. Insurance eligibility, employment screening, and criminal justice sentencing are just some of their operational areas. Since they are deployed in such important fields of society, these algorithms have to take decisions or suggestions for decisions properly to guarantee fair and balanced outcomes. However, subsequent analyses of deployed algorithms have uncovered cases of significant bias towards some demographic subgroups.
This thesis presents an approach to monitor the fairness of an algorithm during its execution. Translating fairness conditions into an RTLola specification makes it possible to generate the monitor. Then, the monitor allows us to check in real-time whether the fairness criteria are fulfilled during an execution of a certain algorithm or not. Furthermore, we investigate the practical feasibility of the approach by comparing several different algorithms for the example of assigning seminars to students.

Zusammenfassung

In unserem Alltag begegnen wir immer öfter Algorithmen, welche uns oder anderen Entscheidungen abnehmen. Versicherungsfähigkeit, Einstellungstests bei Arbeitgebern und strafrechtliche Verurteilung sind nur ein Bruchteil ihrer Einsatzgebiete. Da sie in wichtigen Gebieten der Gesellschaft eingesetzt werden, ist es notwendig, dass diese Algorithmen angemessen entscheiden oder Entscheidungsvorschläge treffen, um ein ausgeglichenes und faires Ergebnis zu erzielen. Jedoch wurde mithilfe von Verfahren,die dem Maschine Learning zuzuordnen sind festgestellt, dass einige Entscheidungsalgorithmen bezüglich demographischer Untergruppen Voreingenommenheit reflektiern, was zu einem unfairen Ergebnis führt.
Unter Verwendung von RTLola als Spezifikationssprache wird in dieser Arbeit die Möglichkeit, Entscheidungsalgorithmen bezüglich Fairness zu untersuchen, vorgestellt. Indem man gewisse Fairness-Eigenschaften zu einer Spezifikation übersetzt, kann man einen Monitor generieren, welchen man für das Untersuchen benötigt. Dies erlaubt es uns zu überprüfen, ob diese Fairness-Kriterien bei bestimmten Algorithmen erfüllt sind oder nicht. Darüber hinaus werden wir das Verhalten der verschiedenen Algorithmen interpretieren und bewerten diese anhand der Ausdruckskraft der Fairness-Eigenschaften. Als Beispiel verwenden wir die Zuordnung von Studenten zu Seminaren.

Contents

Chapter 1
Introduction

Over the past decades, algorithms were established more and more in decision-making procedures. They work whenever they are needed and they also make decisions in much less time than a human. In our everyday life, recommendations and suggestions that are based on an algorithm play a major role. They suggest to us what products to buy or which movie to watch. Furthermore, they are applied in critical situations like hiring decisions [7] or loans [38] .
However, deciding if algorithms determine their result in a "fair" way is still an open question. In practice, algorithms can be "unfair", meaning they discriminate against certain groups of people. Since the data these algorithms are trained on is based on the past, they learn historical biases. This leads to an output that lacks fairness. A popular example is the Correctional Offender Management Profiling for Alternative Sanctions (COMPAS) tool used by courts in the United States. The COMPAS tool is software that calculates the risk of a person committing another crime. Based on this measurement Judges decide whether to release a person or to keep the person in prison. However, analysts found a bias against black defendants in the COMPAS tool. African-American defendants were far more likely determined to be at high risk than white defendants [24].
To take action against this form of bias different forms of fairness conditions have been proposed. However, because of the variety of those conditions, with every condition covering up a specific fairness property, the results differ regarding the fairness aspect. This depends on the definition that is used to check.
The goal of this thesis is to have a tool that reveals algorithms behaving unfairly. To be able to realize this, a strategy that can detect unfairness based on the input of the algorithm and its corresponding output is required. The aim is to use monitoring as the tool that detects unfairness in algorithms. With monitoring, it is possible to check whether an execution of a system satisfies a correctness property. By replacing the correctness property with a fairness condition the monitor can spot unfairness. The input is an execution of a system, and the monitor checks whether the execution satisfies the fairness condition. In addition, it is challenging to have a temporal input, e.g. the input expands as time progresses. With the input not being fixed the monitor still can detect unfairness, even though the input could expand at a later point in time. This opens up the possibility of to model e.g. a seminar assignment system, where students can apply for a place in a seminar. Since everyone can apply at any point in time, up to a certain

deadline, the temporal aspect of the input becomes important.
To be able to model different candidates at different time points we use RTLola, a specification language based on getting input "streams" and features the option of monitoring. These streams are suitable for modeling our input since the elapse of time is part of the stream and can therefore be modeled.
This thesis aims to provide a case study for RTLola where, given several different decision-making algorithms, it detects which of these algorithms are regarding under which fairness condition is considered "fair". The case study focuses on seminar-assignment systems, i.e., given arbitrary input streams of applicants and a decision-making algorithm, the benchmark can detect unfairness under certain conditions. Finally, the monitor is given the input streams and the output. According to that, the monitor detects if the output lies in the fairness condition. To realize the monitor that exposes unfairness the fairness condition is specified in RTLola. It is possible to plug in different algorithms to see the behavior of these algorithms according to fairness.
With that, the goal of monitoring temporal inputs can be realized for inputs given at several stages of time.

Chapter 2
Preliminaries

In this chapter, we briefly introduce the required background on fairness as well as present the specific fairness conditions used in this thesis. Furthermore, we give the required background to the specification language RTLola, which is used to implement these fairness conditions and to continue by monitoring these conditions.
Lastly, this chapter covers the general idea of monitoring and its evaluation.
The following sections introduce fairness in algorithms:

2.1 Algorithmic Fairness

Algorithmic fairness studies the behavior of algorithms regarding biases. In the following sections, we will first see the environment this thesis is based on, i.e. fairness in algorithms is studied in a specific setting in this thesis (2.1.1). After that, a brief background on the origin of fairness definitions is given (2.1.2). Finally, fairness definitions that are required in this thesis, are introduced in Section 2.1.3.

2.1.1 Setting

This thesis is based on the setting of a seminar-assignment system regarding practical experiments. Specifically, this means that within that setting, people can apply at any point in time (up to a certain deadline) to any sort of seminar. Also, every person has an average grade from previous courses, since some factors of applicants are needed to evaluate certain fairness conditions. Eventually, the applicants get a response about whether they are accepted or rejected for the seminar they applied for. Each person can only apply once and also just for one specific seminar.
The goal thereby is detecting unfairness in relation to the gender of applicants. An applicant therefore could look like Figure 2.1

In this case, a woman applied for the computer science seminar. She has an average grade of 2.7 and is accepted to that seminar.
Note that in later examples redundant parameters for the specific fairness conditions are

Figure 2.1: Single Applicant

not shown to keep the examples as simple as possible.

2.1.2 Sources of Unfairness

Bias in algorithms occurs in different shapes and forms. Mostly this happens during the training phase of a machine learning algorithm. However, since a bias does not occur out of nowhere, biases are the result of mistakes during the training process.
The reasons for algorithms behaving unfairly are explained in the following:

Historical Bias
This bias occurs if the data set given to the algorithm during the training phase contains a bias [48]. Consider a data set that contains data from the past, where it was more likely to have a bias against certain demographic groups like people with black skin color. The algorithm might include this historical bias in its behavior [43].

Algorithmic Bias
In difference to historical bias, where the input is the misleading point, algorithmic bias occurs if the algorithm itself behaves biased [4]. This happens when e.g. the algorithm relies on randomization and the random generation function is not purely random. This leads to a bias which then could lead to unfair behavior. Also, algorithmic design choices, such as the usage of optimization functions, regularization, and choices in applying regression models on the data as a whole can all contribute to biased algorithmic behaviour [15].

Evaluation Bias
Evaluation bias happens during model evaluation [43]. The usage of inappropriate benchmarks for evaluation purposes leads to distorted, biased results. E.g. Adience and IJB-A benchmarks are biased against skin color and gender [9]. By using benchmarks like these the outcome will be biased during the evaluation phase.

2.1.3 Fairness Definitions

Considering fairness in algorithms, there does not exist one perfect condition that represents the whole variety of different fairness approaches. In practice, several definitions differ in their expressiveness and targets. According to Kleinberg et al. [28] it is even impossible to satisfy some fairness conditions at once by showing inherent incompatibility of any two out of several conditions. In this thesis, we focus on three popular fairness conditions and study the degree of fairness in algorithms based on these conditions. These conditions are picked since they are not defined for prediction tasks but for tasks with a binary outcome, e.g. true or false.
In the following, the three definitions this thesis is focused, on are expounded:

Definition 2.1.1. *Demographic Parity:*

$$\Big|P[A = 1|G = 1] - P[A = 1|G = 0]\Big| \leq \epsilon$$

Having a lower value of this measure indicates more similar acceptance rates. Therefore, this leads to better fairness. Formula-wise the value of G represents the group the person belongs to (e.g. male or female), while A indicates the positive outcome. ϵ represents the maximum deviation of acceptance rates for each group and has to be specified a priori as a small positive number, e.g. $\epsilon = 10^{-1}$. With that, the absolute value of the difference between these two probabilities should be lower or equal to ϵ. Also, ϵ has to be a positive number). Demographic parity ensures that positive outcomes are assigned to the two groups at a similar rate. Verma and Rubin [47] describe demographic parity as the likelihood of a positive output being the same, no matter if the person is in the protected or unprotected group [31].
Regarding the seminar-assignment system, the two groups (protected and unprotected) correspond to males and females applying to a seminar. Concluding, this means that demographic parity checks whether the acceptance rates for males and females are the same with a certainly acceptable deviation independent of other factors like what seminar they applied for.
Consider the following example:

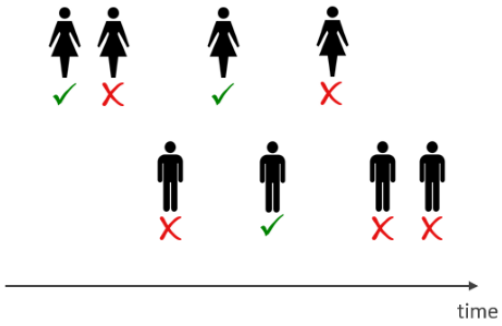


Figure 2.2: Applicantion Scenario

Example 2.1.1. Say that in total eight people apply for any type of seminar as shown in Figure 2.2. Four of them being male and also four being female. Also, Figure 2.2 shows that two of the four females are accepted while only one of the four males is accepted to the applied seminar. Regarding Definition 2.1.1 of demographic parity, we now need to compute the probability of being accepted for both males and females. Trivially for females, it is $\frac{2}{4}$ in this case since two out of four females got accepted to a seminar they applied for, so the probability for a female person being accepted is $\frac{1}{2}$. Since only one of the four males got accepted the probability of getting accepted as a male person calculates to $\frac{1}{4}$. Lastly, we subtract one probability from the other and compute the total number of it, so: $\left|\frac{1}{2} - \frac{1}{4}\right| = \frac{1}{4}$. If $\frac{1}{4}$ is bigger than the value of ϵ this scenario would be unfair according to demographic parity.

Definition 2.1.2. *Conditional Statistical Fairness:*

$$\Big|P[A = 1|L = 1, G = 1] - P[A = 1|L = 1, G = 0]\Big| \leq \epsilon$$

Conditional Statistical Fairness states, similar to demographic parity, that both people in protected and unprotected (female and male) groups should have an equal probability of

positive output. As in demographic parity, G is the value representing the demographic group the person belongs to(male or female), and A is the value for the status of the outcome. In addition to that other legitimate factors are concerned as well [12]. This is represented in the definition by the factor L, which represents the set of factors that are concerned.
For the seminar-assignment setting, this means that the factor of what seminar people apply for now plays a role. With conditional statistical parity, the acceptance rates of both groups are compared within each seminar (e.g. in each subgroup) to detect decision unfairness within specific seminars.
This procedure will be outlined in the following example:

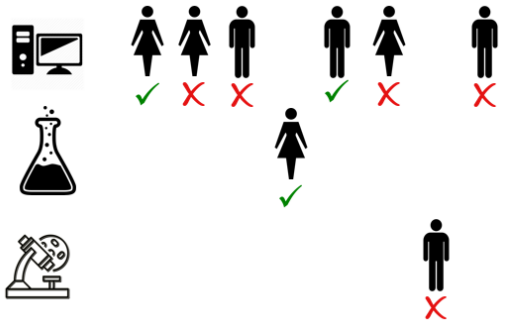


Figure 2.3: Applicants

Example 2.1.2. Again, we have in total eight people applying for seminars. This time, it is specified which seminar they applied for. In this Example (Figure 2.3), six people apply for the computer science seminar with three of them being male as well as three being female. For each other seminar (chemistry seminar and biology seminar) only one person applied for it. In this case, no computations can be done since there are not both demographic groups represented within one seminar. Regarding acceptance rates inside the computer science seminar on average every third woman gets accepted($\frac{1}{3}$). The same counts for males. Concluding this means that if we compute the general acceptance rate according to Definition 2.1.2, we get: $\left|\frac{1}{3} - \frac{1}{3}\right| = 0$. Since $0 \leq \epsilon$, this scenario is definitely fair according to conditional statistical parity.

Simpson's Paradox Example 2.1.1 and Example 2.1.2 showed two scenarios of an application process. Further, in Example 2.1.1 the applied seminar was not regarded and the scenario could be declared as unfair according to demographic parity. However, taking the applied seminar into account as shown in Example 2.1.2, the scenario that is the same regarding the number and gender of applicants is fair.
Having one scenario that is unfair according to demographic parity and fair according to statistical conditional parity is an occurrence of Simpson's Paradox. This paradox states that the result of an investigation over an entire critical group may differ from those of the underlying subgroups.
Since it is possible to compute both results using demographic parity and conditional statistical parity, such a Simpson's paradox can be detected.

Definition 2.1.3. *Counterfactual Fairness:*

$$\left|P[Y_{A\leftarrow a}|S = s, G = 1] - P[Y_{A\leftarrow a'}|S = s, G = 0]\right| \leq \epsilon$$

Counterfactual fairness, also known as Individual fairness, states that two similar individuals are treated similarly [17, 25, 31]. This means that any two individuals who are similar with respect to a similarity get a similar output. In this case, $Y_{A \leftarrow a}$ means that a can be any value attainable by A. Particular this means that every possible outcome should have equal probabilities. L is a set of similarities between two individuals. Lastly, G is the demographic group the person belongs to, and besides that the only difference between the two individuals.
In the set of the seminar assignment system, this means that people that only differ in their gender should have an equal outcome, i.e. should both get accepted or rejected.

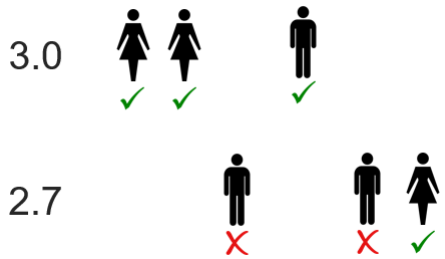


Figure 2.4: Applicants

Example 2.1.3. In this Example (Figure 2.4) the similarity is the average grade of previous courses. We see three people applying for a seminar with an average grade being 3.0. Also, both demographic groups have a similar outcome, namely being accepted. According to counterfactual fairness, this is a fair situation. Taking a look at the people applying with an average grade of 2.7 shows that the males got rejected while the females got accepted. This behavior is, according to counterfactual fairness, unfair.

2.1.4 Estimating Probabilities

We will now cover the problem with its corresponding solution of computing statistically accurate values regarding fairness. In our seminar-assignment environment, the problem of distorted values occurs at the early stages of the monitoring process, i.e. with low numbers of applicants.
A solution for that is "Maximum a Posteriori Probability" (MAP). The intention of this algorithm is to add imaginary, prior information. The algorithm is defined as:

Definition 2.1.4. *Maximum A Posteriori Probability*
In this definition, α_1 describes the number of positive results observed, and α_0 is the number of negative results observed. Further, γ_1 is the number of imaginary positive results, and γ_0 is the number of imaginary negative results. In this environment, "imaginary" refers to the prior value that is artificially added to the actual observation.

$$\theta = \frac{(\alpha_1 + \gamma_1)}{(\alpha_1 + \gamma_1) + (\alpha_0 + \gamma_0)}$$

With that, θ represents the relative frequency of observed positives plus the relative frequency of imaginary positives.

Example 2.1.4. If we use MAP on a coinflip scenario, say "head" is the positive result while "tail" is the negative result. If we now have somehow a reason to assume that θ =

0.6, we can add imaginary flips like $\gamma_1 = 6$ and $\gamma_0 = 4$. With that, at the beginning of the observed coinflips, the estimation θ is at 0.6. Also, a single observed coinflip has now a smaller impact on the general head/tail ratio.

By adapting this to the computation of the acceptance ratio of males and females, it is now possible to execute small examples in our seminar-assignment setting, since a single decision now does not have that big of an impact on the overall ratio. By that, we make the monitor robust by dynamically adapting the metadata before the actual execution of the input.
Concluding, the fairness specifications in RTLola feature an adapted form of MAP that allows us to monitor small groups of applicants making the monitor robust, as well as having statistically accurate results.

2.2 RTLola

We use RTLola [18] in this thesis to describe fairness properties appropriately. RTLola is a stream-based specification language featuring the possibility of monitoring based on real-time computations. Directly, stream-based means that each input type is defined as a separate input stream. Every time an input gets a new value, the corresponding stream gets updated to that value. Computations with these input streams can be done by defining output streams, that act exactly at that time when their needed input streams get updated.

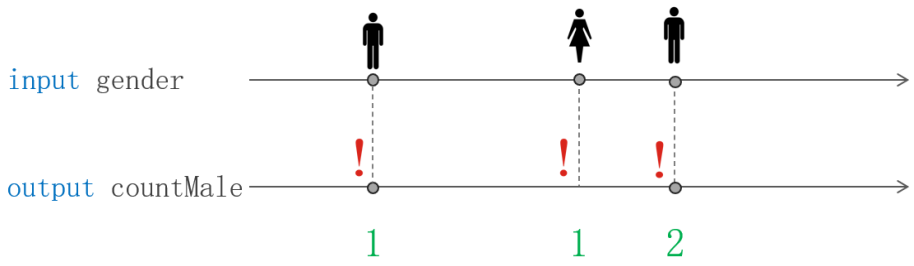


Figure 2.5: Temporal Streams

Example 2.2.1. Let us say that in the seminar-assignment-setting, people apply at different times to seminars. The goal here is to compute the number of male applicants. To represent this, we need an input stream in RTLola. The input stream only focuses on the applicants, i.e. whenever a person applies to a seminar, the value of this stream gets updated to the gender of the newly applied person. This is why we call that input stream "gender", as shown in Figure 2.5. Further, we need an output stream that counts the number of males applying for a seminar. This stream synchronously accesses the input stream always when there occurs a new value in that input stream since there might apply a new male person. By accessing the input stream, the output stream checks whether the new value of the input stream is male, and if so, it counts up the total number of males by one.
Though, if the new value in the input stream is female, the output stream still accesses the stream synchronously and checks for the male gender. However, since the condition is not fulfilled the counter does not get counted up by one but just stays with its value.

Besides that, RTLola introduces asynchronous monitoring to the specification language Lola [14]. In the following section, we first have an overview of the syntax and semantics of RTLola. Thereafter, we describe the transformation from concrete syntax to an abstract syntax tree (AST). (Remark that we only present the required subset of the RTLola grammar.)

2.2.1 Syntax & Semantics

The semantics of RTLola is based on the abstract representation of specification. Therefore, the function *AST* transforms the concrete syntax of a specification to an abstract syntax tree (AST).
In the following table the concrete and abstract syntax for input streams, output streams, and triggers are given and described.
The i^{th} input stream s_i is defined by:

`input i: T`	defines an input stream i of type T

In RTLola, T can be of the following types: Int, UInt, Float, and Bool.
The j^{th} output stream s_j is defined by:

`output o: T`
`eval @i when cond`
`with e`
	defines an output stream o of type T which is described by the expression e and gets evaluated when an input i comes in and *cond* is fulfilled

This syntax does not differ that much from input streams, however, it introduces new components: the keyword "eval", the @i, the condition "cond" as well as the evaluation e. This line of code expresses explicitly that output o is updated when a new value in input stream i occurs and the condition related to i is fulfilled with the value e.

Example 2.2.2. If we look at Figure 2.1, we saw the output stream "countMale", which counts the total number of male applicants. With this definition of output streams, we can realize this output stream. By specifying the evaluation time with `@gender` countMale always accesses the input stream gender synchronously when it gets updated, e.g. a new person applied for a seminar. Implementing the condition `gender == male` ensures that we only execute the expression e when the updated value is male. If this condition is fulfilled, the expression e can be executed which simply counts up the total number of male applicants by one.

The k^{th} trigger s_k is defined by:

`trigger cond "msg"`	warns if the condition *cond* evaluates to false

Trigger contain a condition *cond*, however, a condition can just be any (input-/output-)stream. If the condition is fulfilled the trigger gets activated with the message "msg".

2.2.2 Stream Access

There are several different computation rules for output streams provided in RTLola. To access previous values of a stream, the offset expression is featured in RTLola:

`a.offset(by: n)`	accesses value with offset n of stream a(n has to be negative)

An expression of type: `a.offset (by: n)` is a lookup to access the n^{th} previous value of the stream. This explains the necessity of n being a negative value since it is not possible to access future values. The time the accessor stream looks up this value is bounded by the schedule of the accessed stream.
In case a lookup fails, e.g. the specified offset doesn't exist it is possible to return a desired value via the default command:

`a.defaults(to: v)`	this defines the default value v of stream a(in case it is used out of bounds)

"hold" is defined by:

`a.hold()`	Lookup the current value of a asynchronously

The expression `a.hold()` is used to achieve the accessor stream not being bound to the schedule of the assessed stream. To execute a synchronous lookup it is enough to name the accessed stream:

`a`	Lookup the current value of a synchronously

2.2.3 Parameterization

RTLola also features the possibility to evaluate over parameterized streams. This is desired when e.g. the different values in a stream are arbitrary constants and the goal is to calculate over every different value in that stream. In this case, it is not possible to create an output stream for every value in that stream, since the exact number is not known. Instead, it is possible to create a new output stream for every new value coming into the stream. In this procedure, the output streams are not bounded and therefore it doesn't matter how many different values exist in the stream since the parameterized stream can always create a new output stream if there occurs a new value in the specified stream.
To create a specified stream following expressions are used:

`output o(p)` `spawn with i`	spawns a parameterized stream over i

Having parameterized streams, it is essential to specify over which parameter the stream is parameterized as well as the time the stream starts to live. This can be achieved using

the keyword `spawn`. Following the keyword, it is necessary to indicate at which stream to start living. However, experience shows that this is often the same stream as the one to parameterize.
The evaluation of a parameterized stream can be done the same way as with normal streams.
To close a parameterized stream the following expression is used:

`close cond`	ends the output if condition *cond* is fulfilled

Once a stream is closed it is impossible to gain access to the values evaluated in that stream.

`trigger(p) cond`	with p being an input stream, the trigger over p warns if condition *cond* evaluates to false

Trigger work almost the same as not parameterized triggers with the exception that the parameterized value is specified. This results in a trigger that gets triggered if the condition *cond* is fulfilled in at least one of the different values of the stream that is parameterized over.

2.2.4 Dependency Graph

Since the goal is to build a monitor out of the specification, an analysis of dependency graphs (DG) is required to define semantical validity. RTLola makes use of the definition of DG to describe dependencies between streams. Furthermore, the DG is then used for checking type analysis or the existence of an evaluation model. A dependency graph consists of a finite set of vertices and edges with each vertex representing a stream. To express the relation between the timelines of streams, edges are labeled with the kind of dependency, e.g. if the access occurs in the spawn or close clause, semantic filter, or stream expression. The weight of an edge is the offset to the accessed stream.

Definition 2.2.1. *Dependency Graph:*
The *dependency graph* $\mathcal{G} = (V, E)$ is a labeled graph with the vertices representing the streams and each edge being a dependency

In this context, V is the number of streams in the specification. E represents all dependencies between streams: if and only if s_1 accesses s_2, there exists an edge from s_1 to s_2 in the DG. From that, it follows that, if s_1 accesses s_2 directly or indirectly by accessing another stream before this stream accesses s_2, there is a path from s_1 to s_2 in the DG. To get a brief impression of transforming specification into a dependency graph, consider the following part of an RTLola specification:

```
input gender : String
input seminar : Int64
input accepted : Bool

trigger seminar < 0 // A Seminar is represented as positive Integer

output countWomen
   eval when gender == "F"
```

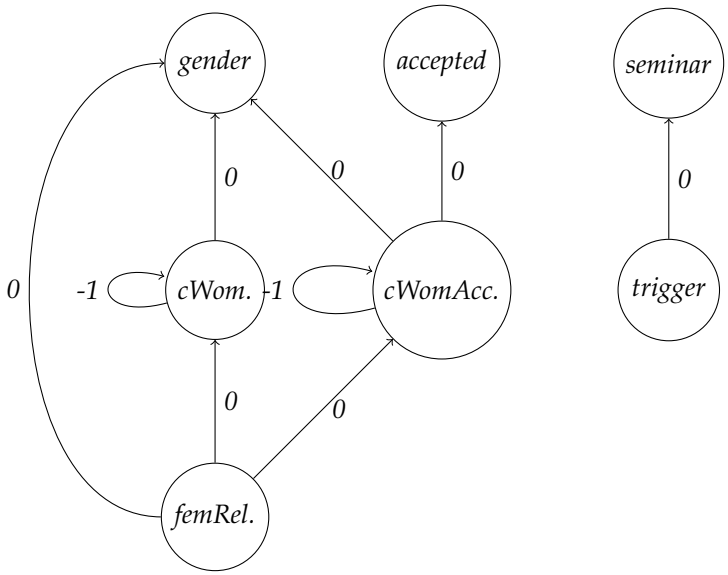
```
    with countWomen.offset(by: -1).defaults(to: 0) + 1

output countWomenAccepted
    eval when accepted == true && gender == "F"
    with countWomenAccepted.offset(by: -1).defaults(to: 0) + 1

output femaleRelation
    eval when gender == "F"
    with (countWomenAccepted.hold(or: 0)) / (countWomen)
```

This specification represents a small seminar assignment system. With the inputs being the gender of an applicant, which seminar the person is applying to, and whether the person is accepted at the seminar or not, at first, the trigger checks whether the seminar is an invalid input. After that, the output stream *countWomen* computes the absolute number of females applying to a seminar. To compute the ratio between women being accepted and all women applying, it is necessary to compute the number of females being accepted.
These computations can then be used to, e.g. check if the relation of accepted individuals between men and women is roughly the same. This proceeding is used later in this thesis to evaluate the accuracy of demographic parity in decision-making algorithms.

Example 2.2.3. This specification has the following dependency graph:



This DG is then used to check for valid execution, i.e. ensure the correct execution order. If this procedure is done then there cannot occur any contradictions regarding the execution process.

2.2.5 Monitoring

In the last section, we saw the syntax and semantics of RTLola as well as the process for checking for semantical validity via dependency graphs(DG). We now briefly introduce the process of monitoring:

Internally, RTLola converts a given specification into a monitor. The procedure of synchronous monitoring operates as follows: the monitor iterates (this may happen infinitely often) over the defined input streams. By getting new input values the monitor updates the values according to the values within the input streams.
In this context synchronous means that all input streams receive an input simultaneously, e.g. all input streams get new values at the same time. In contrast to that, asynchronous monitoring allows input streams to receive values at different times. With that, these input streams are no longer bounded to each other and thus can still be monitored. The number of already analyzed iterations is called *evaluation depth*. According to that value, the monitor iterates over every stream as often as the values of the evaluation depth. Hence, every stream is evaluated as often as the evaluation depth. Therefore each iteration is called *evaluation step*.

Chapter 3
Specifying

In this chapter, we translate the fairness conditions specified in Section 2.1.3 from formulas to equivalent RTLola code. However, we have to deal with the fact that initially, fairness conditions are liveness properties while the resulting RTLola specification has to be a safety property in advance. The goal thereby is to transform these fairness conditions into a similar safety property. This transformed safety property is then translated into RTLola code. To realize this, parameterization in RTLola (Section 2.2.3) is required. Parameterization in RTLola also includes the possibility to investigate more realistic inputs and handle a wider variety of them.
Finally, we see a way to detect potential occurrences of Simpson's paradox by looking at the results from different fairness properties from the same input. This is an important task concerning fairness, since having an occurrence of Simpson's paradox in the data, could lead to mislead evaluation and hence to the wrong conclusion. This is caused by the fact that within Simpson's paradox, the results differ from those within certain subgroups.
In consequence, the RTLola benchmark presented in this thesis detects possible unfairness, while also ensuring that there are no false conclusions.

3.1 Using Parameterization

In the following section, we introduce the naive translation approach from fairness definition to RTLola specification. Resulting from that, we acknowledge that, in order to process more realistic inputs, the usage of parameterization is required. Finally, we see an example of parameterized RTLola specification to understand the necessity of parameterization.

3.1.1 Naive Translation Approach

We now present the naive idea of translating a fairness condition into an RTLola specification code. For this, demographic parity (Definition 2.1.1) is the desired translated

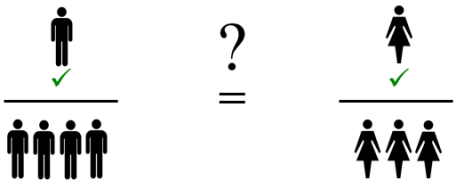


Figure 3.1: Roadmap for demographic parity

fairness definition. However, before we can start to translate, we first need to identify the required values to calculate the acceptance rates for both demographic groups. In this thesis, the represented demographic groups are males and females. Figure 3.1 shows the approach of how to acquire the acceptance rates.

In Figure 3.1 the total values of male applicants and female applicants are required. Furthermore, we need the number of males and females that are accepted to any seminar. We then divide these two values within the demographic group. Lastly, we check whether these two values are similar to each other in order to say if there exists a bias or not according to demographic parity.
We now proceed with the translation process. First, we compute the total number of male and female applicants:

```
input gender : String
input decision : Bool

output countMale
   eval @gender when gender == "M"
   with countMale.offset(by: -1).defaults(to: 0) + 1

output countFemale
   eval @gender when gender == "F"
   with countFemale.offset(by: -1).defaults(to: 0) + 1
```

For this fairness definition, it is enough to have the values of the demographic group, as well as the status of acceptance to a seminar. This results in having the two input streams "gender", where each value is a string ("M" for male, "F" for female) and "decision" as a stream of type boolean (true for accepted, false for rejected). Besides that, we need an output stream that counts the total number of males that apply for a seminar. The output stream "countMale" always checks for updating its value when there occurs a new value in the input stream gender. Further, the condition that the applied person has to be a male person is required. If this condition is fulfilled, the output stream takes its value from the previous iteration (if it exists) and increments by one. This is achieved by having a synchronous lookup to the past iteration of its stream. If there is no previous iteration, e.g. the first person applied to a seminar, we take the default value of zero and add one to it.
For the female people, we compute the exact same with the single difference, that the condition checks, if the person currently applying is a female person.
The following code provides the required number of accepted males and females separately. Note, that we assume that the input streams are still the same as in the code above:

```
output countMaleAccepted
   eval when gender == "M" ∧ decision == true
   with countMaleAccepted.offset(by: -1).defaults(to: 0) + 1

output countFemaleAccepted
   eval when gender == "F" ∧ decision == true
   with countFemaleAccepted.offset(by: -1).defaults(to: 0) + 1
```

These two output streams mostly correspond to the output streams that count the total number of male/female applicants with the major difference of a second condition that has to be fulfilled to update the stream value. By adding the second condition *decision == true*, we ensure that the counter only adds plus one to itself if the applicant got accepted to a seminar. Further, we recognize that it is not explicitly specified when to try to update the value (via an *"eval @"* -statement) since it is not required to indicate the evaluation trigger in RTLola. For readability purposes, we will not provide the explicit evaluation trigger in further RTLola code examples.
Having both necessary values for computing the ratios of each demographic group, the RTLola code shown below contains two new output streams, each containing the current ratio of acceptance of one demographic group:

```
output maleRatio
   eval when gender == "M" with countMaleAccepted.hold(or: 0) / countMale

output FemaleRatio
   eval when gender == "F" with countFemaleAccepted.hold(or: 0) / countFemale
```

Every output stream in the specification code shown above calculates the intended division to gain the acceptance ratio of the corresponding group. It is essential to mention the asynchronous *hold* lookup in the numerator of the fraction. This *hold* means that if *countFemaleAccepted/countMaleAccepted* is not updated in the current iteration of the runtime, the most recent value in *countFemaleAccepted/countMaleAccepted* gets chosen, i.e. *hold* takes the result value of the last successful iteration of *countFemaleAccepted/countMaleAccepted*. We now saw the necessity of the asynchronous *hold* lookup, let us now analyze the reason why it can happen that both of the *"..."-Accepted* streams do not get updated simultaneously with the other output streams.
It is safe to say that we want the rate of acceptance updated always when at least one value of either the numerator or the denominator gets updated. Therefore it is enough to specify the one condition that ensures the correct gender for the respective acceptance ratio. Since the corresponding output streams that compute the total number of applicants of each gender contain the exact same condition, these output streams get updated simultaneously. However, *countMaleAccepted* and *countFemaleAccepted* contains more conditions that have to be fulfilled in order to update their stream values. If there is now a scenario where the condition for the gender is fulfilled but the condition for the decision is not fulfilled, the denominator of the fraction in the ratio streams gets updated while the numerator does not get updated and therefore does not get updated simultaneously with the other output streams.
What still remains to be done is the computation of the difference between the calculated acceptance ratios. Also, to gain feedback from the specification, a trigger is required that informs whenever the acceptance ratios differ too much from each other:

```
output demographicParity
   eval with |maleRatio - femaleRatio|

trigger demographicParity > i
```

The output stream *demographicParity* computes the total value of the subtraction of both acceptance rates. The resulting stream value is the difference in the acceptance rates. If the margin is higher than a certain specified value i (i is a float value), the trigger informs that currently there exists an unfairness according to demographic parity. Note that besides not specifying the evaluation time, there is also no condition specified, since we want the value updated whenever at least one of the two streams *maleRatio* or *femaleRatio* gets updated. If that happens, no further condition is required since it is essential to calculate the current margin of acceptance rates.

3.1.2 Necessity of Parameterization

In the previous section, we introduced the first simple form of implementing a fairness condition into RTLola code. However, this approach is not sufficient for analysis purposes, since we are bound by the simplicity of the specification. The decision, of whether a person gets accepted to the applied seminar, has to be made at the time of the appliance. Though, in practice, such decisions may be done after a given deadline. Until that deadline, people can apply to seminars, and when that deadline has been reached, decisions of acceptance are made for all applicants. In the current state of the fairness condition, this could not be monitored, since the fairness condition expects the decision values at the time the people apply for a seminar. To realize working with different forms of delay, the use of parameterization in RTLola is required. With that, we are able to store data from each applicant and look it up at a later point in time, when we need it. Further, as we saw above, it is possible to implement (a simple form of) demographic parity without parameterization. If we take a look at conditional statistical parity (Section 2.1.2) though, we see that computations within each different seminar are required. Since it is not always clear how many seminars are offered, again, parameterization is required. With that, we can realize computations within each seminar. For every different seminar, a new output stream spawns and one can do computations separately.
Concluding, with parameterization we can accomplish monitoring more realistic inputs as well as realizing the implementation of all fairness conditions presented in this thesis.

3.1.3 Parameterization Example

In the following example, we see a snippet of RTLola code using parameterization. We want to count the total amount of applicants within each seminar. Since it may not be clear in all scenarios how many different seminars there exist, we use parameterization, since we can handle arbitrary numbers of seminars with that. The parameterized specification that counts the number of applicants for each seminar is shown below:

```
input gender : String
input seminar : Int64
input decision : Bool

output countApplicants(p)
   spawn with seminar
```

```
eval when seminar == p
with countApplicants(seminar).offset(by: -1).defaults(to: 0) + 1
close when countApplicants(seminar) ≥ 12
```

In the specification fragment, we see the input streams *gender*, *seminar*, and *decision*. Note, that in this case we only need the information of the *seminar* stream, since it is enough to calculate the number of applicants within each seminar. The output stream *countApplicants* is parameterized, indicated from the *(p)*. The spawning condition is over the seminars. Specifically, this means that the stream spawns a new separate output stream for each new value in the input stream *seminar*. The stream evaluates a new value every time when there is a new value in the input stream *seminar*. Then, the corresponding output stream from the parameterization gets updated with the value from the previous iteration plus one. When there are 12 or more applications for the same seminar, the corresponding parameterized stream closes. With that, it is possible to see in which seminars are still spaces left.
Using this technique, it is possible to evaluate fairness over each type of seminar, as well as parameterize over each applicant, resulting in stored data from each user. With this approach, we can compute the desired fairness conditions in this thesis using specifications in RTLola.

3.2 Fairness Conditions as Liveness Properties

In this chapter, we analyze the behavior of fairness conditions as liveness properties and how we can form a new safety property that represents the same expression as the initial fairness condition. Therefore, we first introduce safety and liveness properties themselves and then transfer this definition to the respective fairness conditions. With that, we present the transforming method for the fairness conditions used in this thesis from liveness property to safety property in order to utilize them later in this thesis.

Liveness to Safety Properties Liveness properties ensure that "good things do happen" [32]. This means directly that something has to happen during the execution of a program code. Further, the "good thing" that has to happen, does not need to be discrete. With that, it could require an infinite number of steps, since no matter how the program execution starts, it is always possible to "save" the execution by ensuring that the good thing will happen eventually in the future. However, most of the time it is not satisfactory to know that the program eventually does a "good thing". Though, giving a specific bound to the "good thing" would result in having a safety property rather than remaining the liveness property.
Transferring this property to fairness conditions, we see that we can adapt the definition of liveness property to the fairness conditions: the "good thing", in the fairness scenario, is that ultimately, the fairness condition does not get harmed. Also, no matter how "bad" the algorithm starts to decide (by e.g. accepting one hundred females but zero males) it is always possible to save the current execution (by accepting only one hundred males from now on) with a correct suffix from the current state of execution. Eventually, the algorithm has to be fair, but the algorithm is not unfair by having one point in time where the execution yields unfairness, since it always can be fixed in the future.
Since we cannot monitor an infinite execution, we need to transform the initial fairness conditions to properties with a specific bound, which makes them safety properties.

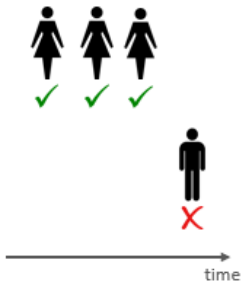


Figure 3.2: Small input Example

A safety property proscribes "bad things" from occurring in an execution [32]. With that, the "bad thing" is discrete, i.e. if there is a "bad thing", it has to occur at some identifiable point. If a "bad thing" happened, it is impossible to fix the execution with a suffix for the execution.
Concluding, we need to introduce some bounds to the fairness conditions in order to get a bounded property, which then is possible to be monitored since it is now a safety property. Having a specific bound ensures that the monitor never has to analyze an infinite execution sequence.

Besides the necessity of changing the initial fairness conditions in order to be able to monitor them, it has also advantages regarding the relation to the real world. Usually, it is not beneficial if an algorithm performs fair eventually, but rather unfair in realistic bounds. In the real world, it is not sufficient if an algorithm will be fair eventually in the future with a certain number of steps. Rather, this thesis follows the hypothesis that observed unfairness is focused on the past, not on the future. In cases of specific deadlines or time limits, it has the advantage to introduce the bound to the fairness conditions to evaluate the algorithm in more detail and nearer to the real-world example. By looking for prefixes within that bound violating fairness, we can realize the conversion of the fairness conditions. Moreover, with the bound, the initial expression of the fairness conditions does not get changed since we only focus on a certain time window of the execution where we look for bad prefixes. However, it is required to introduce a certain "tolerance" to the fairness conditions, realized by the "$\leq \epsilon$" in the definitions of demographic parity (Section 2.1.1) and conditional statistical parity (Section 2.1.2). This ϵ is the tolerance, meaning that the acceptance rates of males and females do not have to be exactly the same but rather be not too far apart. The tolerance is a constant value, since we do not require a dynamic tolerance when using maximum a posteriori estimation (MAP) (Definition 2.1.4). The following section (Section 3.3) covers the necessity, as well as the usage of MAP in our calculations.

3.3 Using Maximum a Posteriori Estimation

We now show the necessity of using maximum a posteriori probability estimation (MAP) (Section 2.1.4). For that, we first come up with a scenario with a small number of applicants and try to check for unfairness naively. After that, we do the same example, but now in addition with a MAP. Finally, we will discuss the results of both examples.
Say, we want to check for unfairness according to demographic parity (Section 2.1.1) with the input shown in Figure 3.2

In this example (Figure 3.2), all three female applicants get accepted, while the single applicant does not. Note, that in this scenario we do not focus on temporal aspects, so the time of the decision made for these applicants is not of interest. Further, say that all apply for the same seminar.

Naive Approach Computing now the acceptance ratio of both demographic groups (Figure 3.1) yields the following results:
The acceptance rate of females is $\frac{3}{3} = 1.0$ meaning that every woman that applied for a seminar also got accepted. Thus for males, the acceptance rate is $\frac{0}{1} = 0.0$, since the only man that applied got rejected. According to demographic parity, this scenario is highly unfair. By comparing the difference of the acceptance ratios for both demographic groups, we get $|1.0 - 0.0| = 1.0$. However, the maximum deviation that is tolerated by the trigger is a constant, say 0.2.

Using MAP Investigating the same example by additionally using MAP, however, the difference between both acceptance ratios is not that high:
One seminar has a capacity of twelve. Say that for each seminar that is seen so far, six applicants from each demographic applied for a seminar. From both demographic groups, all of them got accepted, resulting in the prior acceptance rate being $\frac{6}{6} = 1.0$ for each demographic group. If we now investigate the input scenario with also applying the formula for MAP (Definition 2.1.4), we get $\theta_F = \frac{3+6}{(3+6)+(0+0)} = \frac{9}{9} = 1.0$ for females and $\theta_M = \frac{0+6}{(0+6)+(1+0)} = \frac{6}{7}$ for males. Comparing these two acceptance rates yields $|1.0 - \frac{6}{7}| \approx 0.143$ which is still in the bound of 0.2.

With MAP we can make the monitor robust, i.e. the monitoring process is statistically accurate, independently of the number of applicants. If an algorithm makes seemingly unfair decisions within its first few steps, it is wrong to say that the whole algorithm is unfair, since after a few decisions it is hard to tell whether an algorithm behaves fair or not in general.
Though, utilizing MAP alone is not sufficient since the prior needs to be normalized fairly. With normalization, we handle cases, where one specific demographic group occurs predominantly. Without normalization, this would mean that the prior from the minor demographic group impacts too much and outvote the actual decisions.
To overcome this problem, we modify the initial definition of MAP (Definition 2.1.4) featuring normalization:

Definition 3.3.1. *New Definition of MAP with Normalization*
Here, γ_1 is the number of imaginary positive results and γ_0 the number of imaginary negative results. Also, α_1 are the actual positive outcomes and α_0 are the actual negative decisions. Further, κ describes the confidence, and o the overflow of applicants (if exists). Hence, o is the difference between applicants and total spaces in the seminars. If this value is smaller than zero, o becomes zero

$$\hat{\theta} = \frac{\alpha_1 + \frac{\gamma_1}{\gamma_1+\gamma_0+o} * \kappa}{\alpha_1 + \alpha_0 + \kappa}$$

With this definition (Definition 3.3.1) we gain the advantage that the confidence parameter stays fixed in the denominator. This results in the acceptance rate not changing rapidly when there are new decisions, and therefore new acceptance rates, for the other demographic group. This is realized by adding the proportion of imaginary positives (γ_1) and all imaginary applicants ($\gamma_1 + \gamma_0$) added to the overflow of actual applicants (o)

in the numerator.
Using this adapted definition, the example (Figure 3.2) yields following results:

MAP plus Normalization In the scenario of Figure 3.2, the acceptance rate for females, with a confidence of six, would be: $\hat{\theta}_F = \frac{3+\frac{6}{6+0+0}*6}{3+0+6} = \frac{9}{9} = 1.0$ while the acceptance rate for males would be $\hat{\theta}_M = \frac{0+\frac{6}{6+0+0}*6}{0+1+6} = \frac{6}{7}$.
Here, we see the same results as in the approach, where we only made use of MAP. Though, this new approach can handle cases, where the number of applicants from the demographic groups differs considerably. Further, this approach features the possibility of adapting the confidence κ to adapt the level of robustness.

3.4 Translating Fairness Conditions Into RTLola Specifications

We now focus on the actual translation and implementation of the fairness definitions. In the last sections, we saw the necessity of parameterization in order to analyze and monitor realistic scenarios. Also, we saw the idea of bounding these fairness definitions as liveness properties to gain safety properties with their expressiveness not being changed.

Remark. In all the following implementations of fairness definitions, we ignore types. In many cases, a cast from an Int to a Float (cast<UInt64,Float64>) is required. however, for reading purposes, we omit this specific cast.

In the following, we will see the actual implementation of fairness conditions in RTLola:

3.4.1 Demographic Parity

We now give the implementation of demographic parity. To be able to check whether the fairness condition to hold, every applicant needs several parameters:

```
input time : Int64
input id : Int64
input gender: String
input seminar: Int64
input qualification: Int64
input accepted: Bool
input idAccepted: Int64
```

The parameters are represented as input streams. Each time a new person applies for a seminar, these streams get updated with their new value. We need a *time* parameter to make statements about the timing of applicants, as well as the *id* as a parameter to uniquely identify each person. The *gender* is required to make calculations about the acceptance rates of the demographic groups and the *seminar* parameter is primarily for conditional statistical parity (Section 2.1.2). Besides that, the *qualification* parameter is mainly used by counterfactual fairness (Section 2.1.3). Both input streams *accepted* and *idAccepted* exist for realizing the temporal aspect, namely the delay of the decision. At a

point later in time than the application, the decision for the person is made. Meaning, that *accepted* is the parameter for the decision and *idAccepted* is to whom the decision belongs.
To generate better readability in the code, we introduce ENUMs' representing the state of the decision:

```
constant UNDEFINED : UInt64 := 0
constant ACCEPTED : UInt64 := 1
constant REJECTED : UInt64 := 2
```

With that, we gain the advantage of having now a third value "UNDEFINED" in addition to the truth values of the Boolean.
The next step is to store the provided data from each applicant. If we are at a time, where a decision for a previous application is made, we only have the decision and the id for the applicant provided in the input table. Though, in order to make calculations about acceptance rates, we need more information, like the gender or the seminar the person applied for. For that, we introduce a triple that stores the gender, the seminar, and the decision of a person:

```
output userData(p)
   spawn with id
   eval when id == p with (gender,seminar)
   close when p == idAccepted

output userDataAccepted(p)
   spawn with idAccepted
   eval when idAccepted == p with if accepted then ACCEPTED else REJECTED
```

This parameterized output stream *userData* spawns a new stream for each new id that occurs in the input stream *id*. This stream then stores the parameters *gender* and *seminar* for the person currently applying. The stored data then is needed later, when this exact person gets a decision, for calculations shown later in this section. Also, the stream for a person is closed when the according person gets a decision. In that timestamp, we calculate all needed values and with that, the stream that stores data from a specific person can be closed then.
Further, *userDataAccepted* also spawns a new stream for every different id that occurs in the stream *idAccepted*. It then stores the decision of the corresponding person, i.e. either "ACCEPTED" or "REJECTED". This results in three parameters stored for each person in total.
Having now the required data stored, we can start to investigate for unfairness according to demographic parity (Definition 2.1.1). For that, we need the total value of males and females that applied for a seminar. We generate a separate output stream for each demographic group:

```
output allMale
   eval @idAccepted when
      userData(idAccepted).hold(or: ("D",0)).0 == "M"
   with allMale.offset(by: -1).defaults(to: 0) + 1

output allFemale
   eval @idAccepted when
```

```
        userData(idAccepted).hold(or: ("D",0)).0 == "F"
    with allFemale.offset(by: -1).defaults(to: 0) + 1
```

These two output streams count up the number of applied males and females that already have a decision of whether they are accepted or not. These streams check at the time of a new value in the *idAccepted* stream if the person with this id is male or female. This is done by accessing the parameterized stream *userData* with the parameter *idAccepted*. With that, we access the tuple with the stored data for the applicant that currently received their decision. Also, the *hold* operator ensures that the stream gets accessed asynchronously if needed. Depending on the gender, the counter is increased by one by taking the value of the stream from the last iteration and adding plus one to it. To compute the acceptance rates, we also need the number of males and females, that are accepted to a seminar separately:

```
output maleAccepted
    eval when userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
        ∧ userData(idAccepted).hold(or: ("D",0)).0 = "M"
    with maleAccepted.offset(by: -1).defaults(to: 0) + 1

output femaleAccepted
    eval when userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
        ∧ userData(idAccepted).hold(or: ("D",0)).0 = "F"
    with femaleAccepted.offset(by: -1).defaults(to: 0) + 1
```

These streams check if the person that currently got their decision is accepted, as well as if the gender of this person is matching. For both parameters, the *userDataAccepted* of this person is checked if the person is indeed accepted. Further, these streams check the *userData* for the gender of the corresponding person. If both conditions are fulfilled, the stream for the according demographic group gets counted up by one.
The next step of the computation is to calculate the acceptance rates for each demographic group. Additionally, we saw earlier that the naive formula for the acceptance rates, namely $\frac{(fe-)maleAccepted}{all(Fe-)Male}$ is not sufficient (Section 3.3). Thus, it is required to use a modified formula for the acceptance rates, that also implements maximum a posteriori estimation (MAP) as seen in Definition 2.1.4. MAP introduces prior, imaginary, decisions that result in one actual decision having less impact on the acceptance rates.
To implement the modified formula for the acceptance rates, the number of imaginary decisions has to be calculated first. The amount of prior, imaginary decisions is dependent on the number of different seminars seen so far in the actual input list. For each new seminar seen, the prior value increases, resulting in single decisions having less impact and acquiring a robust monitor.
In the following, the prior values are calculated:

```
output seminarTicker(p)
    spawn with seminar
    eval when seminar.offset(by: -1).defaults(to: 0) == p with true

output seminarCounter
    eval @seminar with (if seminarTicker(seminar).hold(or: false)
        then seminarCounter.offset(by: -1).defaults(to: 0)
        else seminarCounter.offset(by: -1).defaults(to: 0) + 1)
```

The stream *seminarTicker* is parameterized over the seminars, i.e. generates a new stream for every different seminar seen in *seminar*. The purpose of this stream is to see which seminars are already seen and which are new since the prior value increases for each new seminar. If there is a new seminar, the stream sets the according parameterized stream to true one timestamp later.
To gain the number of different seminars seen so far, *seminarCounter* checks if the seminar currently in the input stream *seminar* has been seen already. This is done by checking for the according parameterized stream whether this stream is true. If the stream is new, *seminarCounter* increases by one. Otherwise, the value stays the same.

```
output overflowCounter
   eval @idAccepted
   with (if (allMale.hold(or: 0) + allFemale.hold(or: 0))
         >= (seminarCounter.hold(or: 0) * 12)
      then (allMale.hold(or: 0) + allFemale.hold(or: 0))
         - (seminarCounter.hold(or: 0) * 12)
      else 0)

output priorValue
   eval @idAccepted
   with (seminarCounter.hold(or: 1) * 12)
      / ((seminarCounter.hold(or: 1) * 12) + overflowCounter)
```

The output stream *overflowCounter* evaluates whenever there is a new decision made and checks whether there applied more people than there are total spaces in the seminars. If this is the case, the stream adapts the value of the difference between the total applicants and total spaces in the seminars, else the value is zero. This value is required for the calculation of the acceptance rates using MAP and normalization (Definition 3.3.1) since we need the number of people that applied more than there are spaces in all seminars.
Then, *priorValue* calculates the total number of spaces in the seminar divides by the total number of spaces plus the overflow, i.e., $\frac{\gamma_1}{\gamma_1+\gamma_0+o}$ from Definition 3.3.1.
Having now calculated the necessary prior values, it is now possible to compute the acceptance rates:

```
output maleRelation
   eval when userData(idAccepted).hold(or: ("D",0)).0 == "M"
   with (maleAccepted.hold(or: 0) + (priorValue.hold(or: 1.0) * 6.0))
      / (allMale + 6)

output femaleRelation
   eval when userData(idAccepted).hold(or: ("D",0)).0 == "F"
   with (femaleAccepted.hold(or: 0) + (priorValue.hold(or: 1.0) * 6.0))
      / cast<UInt64, Float64>(allFemale + 6)
```

The condition in each stream ensures that the acceptance ratios are updated when a person from the corresponding demographic group gets a decision. When the condition is fulfilled, these streams calculate the acceptance rate including MAP and normalization (Definition 3.3.1). Further, *all(Fe-)Male* corresponds to $\alpha_1 + \alpha_0$ in the definition of MAP. Note, that MAP plus normalization is applied separately to every demographic group. These robust acceptance rates can now be compared:

```
output statParity
   eval @idAccepted with abs(maleRelation.hold(or: 0.5) -
        femaleRelation.hold(or: 0.5))
```

The stream *statParity* calculates the absolute value of the difference between both acceptance rates.
The final step is to check whether this value is higher than a certain fixed value:

```
trigger statParity > 0.1
```

The trigger always activates when the discrepancy between acceptance rates is higher than 0.1.

In this section, we saw the actual implementation of demographic parity (Definition 2.1.1). With this implementation, experiments are done later in this thesis (Chapter 5). The next section shows the realization of conditional statistical parity (Definition 2.1.2) in RTLola.

3.4.2 Statistical Conditional Parity

In the last section, we saw the implementation of demographic parity. We now see the implementation of statistical conditional parity step-by-step. Note, that in many cases the implementation of statistical conditional parity is similar to the implementation of demographic parity. The major difference between the implementation of demographic parity and conditional statistical parity is that computations regarding acceptance rates of demographic groups are now calculated separately for every different subgroup.
As for demographic parity, we now also need the following input streams:

```
input time : Int64
input id : Int64
input gender: String
input seminar: Int64
input qualification: Int64
input accepted: Bool
input idAccepted: Int64
```

Since this fairness definition investigates acceptance ratios within seminars, the parameters *gender, seminar* and *accepted* are important in this section.
Also, since there are three different states of the decision, namely accepted, rejected, and undefined, there are the same ENUM's as in demographic parity:

```
constant UNDEFINED : UInt64 := 0
constant ACCEPTED : UInt64 := 1
constant REJECTED : UInt64 := 2
```

We further use the same structure for storing the data from each user as in the previous fairness implementation:

```
output userData(p)
   spawn with id
   eval when id == p with (gender,seminar)
   close when p == idAccepted

output userDataAccepted(p)
   spawn with idAccepted
   eval when idAccepted == p with if accepted then ACCEPTED else REJECTED
```

These two parameterized streams store the gender, the seminar, and the state of the decision for each user. These informations are required for later calculations and therefore need to be stored until the calculation process is over.
With the necessary data stored, it is now possible to start making computations for fairness according to statistical conditional parity (Definition 2.1.2):

```
output allMale(p)
   spawn with seminar
   eval @idAccepted when userData(idAccepted).hold(or: ("D",0)) == ("M",p)
      with allMale(p).offset(by: -1).defaults(to: 0) + 1

output allFemale(p)
   spawn with seminar
   eval @idAccepted when userData(idAccepted).hold(or: ("D",0)) == ("F",p)
      with allFemale(p).offset(by: -1).defaults(to: 0) + 1
```

These two parameterized streams calculate the total number of applicants for each demographic group within each seminar. For each different seminar, these streams spawn a new stream that counts up the number for the new seminar. The condition now also checks which seminar the person applied for. According to that, the corresponding parameterized stream gets counted up by one. Besides that, these two streams behave similarly to *all(Fe-)Male* streams from demographic parity (Section 3.4.1).
For the acceptance rates, we also need the number of accepted applicants for each demographic group within each different seminar:

```
output maleAccepted(p)
   spawn with seminar
   eval when userData(idAccepted).hold(or: ("D",0)) == ("M",p)
        ∧ userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
      with maleAccepted(p).offset(by: -1).defaults(to: 0) + 1

output femaleAccepted(p)
   spawn with seminar
   eval when userData(idAccepted).hold(or: ("D",0)) == ("F",p)
        ∧ userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
      with femaleAccepted(p).offset(by: -1).defaults(to: 0) + 1
```

These two output streams work similarly to *all(Fe-)Male(p)* with the single difference that these streams also check the status of the decision. Only if the person got accepted to a seminar, the corresponding parameterized stream gets counted up by one.
Since we saw that the naive formula for the acceptance rates is not sufficient (Section 3.3), we need further calculations to use MAP and normalization (Definition 3.3.1):

```
output overflowCounter(p)
   spawn with seminar
   eval @idAccepted
      when userData(idAccepted).hold(or: ("D",0)).1 == p
      with if (allMale(p).hold(or: 0) + allFemale(p).hold(or: 0)) >= 12
      then ((allMale(p).hold(or: 0) + allFemale(p).hold(or: 0)) - 12)
      else 0

output priorValue(p)
   spawn with seminar
   eval @idAccepted when userData(idAccepted).hold(or: ("D",0)).1 == p
   with 12.0 / (12.0 + (overflowCounter(p)))
```

The stream *overflowCounter(p)* checks if more people applied than there are spaces for each seminar. If this is the case, the parameterized stream takes the value of the difference of spaces for that seminar and applicants for that seminar. Else, the parameterized stream has a value of zero.
These values are required for the calculations of the *priorValue(p)* stream. This stream calculates the ratio between total spaces and total spaces plus overflow for each different seminar. Hence, this stream calculates $\frac{\gamma_1}{\gamma_1+\gamma_0+o}$ from Definition 3.3.1 for each seminar.
Now that all necessary values are computed, we can calculate the actual acceptance rates for both demographic groups:

```
output maleRelation(p)
   spawn with seminar
   eval when userData(idAccepted).hold(or: ("D",0)) == ("M",p)
       with (maleAccepted(p).hold(or: 0)
          + (priorValue(p).hold(or: 1.0) * 6.0))
          / (allMale(p) + 6)

output femaleRelation(p)
   spawn with seminar
   eval when userData(idAccepted).hold(or: ("D",0)) == ("F",p)
       with (femaleAccepted(p).hold(or: 0)
          + (priorValue(p).hold(or: 1.0) * 6.0))
          / (allFemale(p) + 6)
```

The streams are parameterized since these acceptance rates are computed separately for every different seminar. Every stream checks if the person, that currently got a decision, is of the "correct" demographic group and also, which seminar this person applied for. Then, these streams calculate the acceptance rates according to Definition 3.3.1.
Now these acceptance rates remain to be compared:

```
output condParity(p)
   spawn with seminar
   eval @idAccepted when userData(idAccepted).hold(or: ("D",0)).1 == p
   with abs(maleRelation(p).hold(or: 1.0) - femaleRelation(p).hold(or: 1.0))
```

With *condParity(p)* we get separate values for each different seminar. Each parameterized stream stores the difference between the acceptance rates of both demographic groups within that specific seminar.
Lastly, we need to check if (at least) one of these parameterized values of *condParity(p)*

is too high, and whether there is a too big discrepancy between the acceptance rate for males and females:

```
trigger condParity(userData(idAccepted).hold(or: ("D",0)).1).hold(or: 0.0) >
    0.1
```

The trigger activates always when the discrepancy between both acceptance rates within a specific seminar is higher than 0.1.

We now saw the implementation of both demographic parity (Section 3.4.1) as well as statistical conditional parity (Section 3.4.2).
The next section shows the implementation of counterfactual fairness (Definition 2.1.3).

3.4.3 Counterfactual Fairness

This section covers the implementation of counterfactual fairness (Definition 2.1.3). Note, that the original intention of counterfactual fairness, namely that "a decision is fair towards an individual if it is the same in both the actual world and a counterfactual world where the individual belonged to a different demographic group" [36] is not realizable to implement. For that, it would be required to have a database in RTLola where computations for the "counterfactual world". could be done. Hence, the implementation provided in this section realizes a new interpretation of this fairness definition:

Modified Interpretation Instead of checking whether the decision for an individual would be the same in a counterfactual world, this new approach checks if two applicants within one scenario that only differ in their demographic group get the same result (Example 2.1.3). Further, if a seminar has no spaces left, the excessive applicants do not count for individual unfairness, even though, the original definition of that fairness states so. The reason that case does not count as individual unfairness is the fact that in a temporal approach, it occurs that there apply more people than there are spaces in seminars. Hence, if every excessive applicant would count as an unfair decision, there would exist (almost) no scenario that is fair according to counterfactual fairness.

Remark. Before the actual implementation, we introduce syntactical sugar for readability purposes. By comparing the values of a tuple, a "_" indicates that this value can be any value and therefore will not be regarded.

As for both other fairness conditions, we need the following input streams:

```
input time : Int64
input id : Int64
input gender: String
input seminar: Int64
input qualification: Int64
input accepted: Bool
input idAccepted: Int64
```

Since this fairness definition compares individuals that have the same qualification but differ in the demographic group, the parameters *gender, qualification* and *accepted* are

important in the following implementation.
For the different states of decision, we again use the following ENUM's:

```
constant UNDEFINED : UInt64 := 0
constant ACCEPTED : UInt64 := 1
constant REJECTED : UInt64 := 2
```

There also has to be stored the required data from each applicant, which is realized by similar parameterized output streams as in both other fairness definitions:

```
output userData(p)
   spawn with id
   eval when id == p with (gender,qualification,seminar)
   close when p == idAccepted

output userDataAccepted(p)
   spawn with idAccepted
   eval when idAccepted == p with if accepted then ACCEPTED else REJECTED
```

Note that we now also store the *qualification* parameter in the *userData* stream, since this value is required for later computations.
Then we need to keep track of whether people from each demographic group get accepted with their corresponding qualification:

```
output maleQualificationAccepted(p)
   spawn with qualification
   eval when userData(idAccepted).hold(or: ("D",0,0)) == ("M",p,_)
        && userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
   with true

output femaleQualificationAccepted(p)
   spawn with qualification
   eval when userData(idAccepted).hold(or: ("D",0,0)) == ("F",p,_)
        && userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
   with true
```

These two parameterized streams check for every demographic group whether there is a positive decision made and if this is the case, the stream with the according qualification gets set to true. With that, we know for what qualifications people got accepted to a seminar.
Further, we need the same streams as shown above with the difference to check whether the applicant gets rejected.

```
output maleQualificationRejected(p)
   spawn with qualification
   eval when userData(idAccepted).hold(or: ("D",0,0)) == ("M",p,_)
        && userDataAccepted(idAccepted).hold(or: UNDEFINED) == REJECTED
   with true

output femaleQualificationRejected(p)
   spawn with qualification
   eval when userData(idAccepted).hold(or: ("D",0,0)) == ("F",p,_)
```

```
        && userDataAccepted(idAccepted).hold(or: UNDEFINED) == REJECTED
    with true
```

Since if in a seminar all spaces are occupied all following applicants get denied and therefore, unfairness would arise, we also keep track of the remaining free spaces:

```
output seminarCounter(q)
    spawn with seminar
    eval when userData(idAccepted).hold(or: ("D",0,0)).2 == q
        && userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED
    with seminarCounter(q).offset(by: -1).defaults(to: 0) + 1
```

This parameterized stream counts up the number of accepted applicants for each seminar. With the use of this stream, it is then possible to see whether there are free spaces left in specific seminars.
Since now all required data is stored, it is possible to check for unfair decisions:

```
output unfairCheckerMaleR
    eval with if ((userData(idAccepted).hold(or: ("D",0,0)).0 == "M")
        && (userDataAccepted(idAccepted).hold(or: UNDEFINED) == REJECTED)
        && femaleQualificationAccepted(userData(idAccepted).hold(or:
            ("D",0,0)).1).hold(or: false)
        && seminarCounter(userData(idAccepted).hold(or:
            ("D",0,0)).2).hold(or: 0) <= 11)
        then unfairCheckerMaleR.offset(by: -1).defaults(to: 0) + 1
        else unfairCheckerMaleR.offset(by: -1).defaults(to: 0)

output unfairCheckerFemaleR
    eval with if ((userData(idAccepted).hold(or: ("D",0,0)).0 == "F")
        && (userDataAccepted(idAccepted).hold(or: UNDEFINED) == REJECTED)
        && maleQualificationAccepted(userData(idAccepted).hold(or:
            ("D",0,0)).1).hold(or: false)
        && seminarCounter(userData(idAccepted).hold(or:
            ("D",0,0)).2).hold(or: 0) <= 11)
        then unfairCheckerFemaleR.offset(by: -1).defaults(to: 0) + 1
        else unfairCheckerFemaleR.offset(by: -1).defaults(to: 0)
```

These two streams check for unfairness, each for one demographic group. They always get evaluated when there is a new decision made for an applicant. With that, *unfairCheckerMale* and *unfairCheckerFemale* check, whether the person that currently got their decision belongs to the corresponding demographic group. Further, they investigate whether this person got rejected. If this is the case, these streams look up in *(fe-)maleQualification* if there exist people belonging to the other demographic group who got accepted with the same qualification. Finally, they check if there is still space left in the applied seminar.
Further, the same checks have to be done, except that the decision now has to be accepted and we look at (fe-)maleQualificationRejected.

```
output unfairCheckerMaleA
    eval with if ((userData(idAccepted).hold(or: ("D",0,0)).0 == "M")
        && (userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED)
        && femaleQualificationRejected(userData(idAccepted).hold(or:
            ("D",0,0)).1).hold(or: false)
```

```
        && seminarCounter(userData(idAccepted).hold(or:
            ("D",0,0)).2).hold(or: 0) <= 11)
        then unfairCheckerMaleA.offset(by: -1).defaults(to: 0) + 1
        else unfairCheckerMaleA.offset(by: -1).defaults(to: 0)

output unfairCheckerFemaleA
   eval with if ((userData(idAccepted).hold(or: ("D",0,0)).0 == "F")
        && (userDataAccepted(idAccepted).hold(or: UNDEFINED) == ACCEPTED)
        && maleQualificationRejected(userData(idAccepted).hold(or:
            ("D",0,0)).1).hold(or: false)
        && seminarCounter(userData(idAccepted).hold(or:
            ("D",0,0)).2).hold(or: 0) <= 11)
        then unfairCheckerFemaleA.offset(by: -1).defaults(to: 0) + 1
        else unfairCheckerFemaleA.offset(by: -1).defaults(to: 0)
```

When all these conditions evaluate to be true, individual unfairness is detected, and the stream counts up its value by one. Otherwise, it keeps its former value.
To warn the user, there are triggers that rise when there occurs a case of unfairness:

```
trigger unfairCheckerMaleR
   > unfairCheckerMaleR.offset(by: -1).defaults(to: 0)
trigger unfairCheckerFemaleR
   > unfairCheckerFemaleR.offset(by: -1).defaults(to: 0)
trigger unfairCheckerMaleA
   > unfairCheckerMaleA.offset(by: -1).defaults(to: 0)
trigger unfairCheckerFemaleA
> unfairCheckerFemaleA.offset(by: -1).defaults(to: 0)
```

These triggers check whether the value of either *unfairCheckerMale* or *unfairCheckerFemale* got counted up by one. If this is the case, a trigger is thrown.

Summarizing, this section showed the implementation of fairness definitions, that are investigated in this thesis.
The following section covers the input generation that is used to realize later experiments.

Chapter 4
Input Generation

This section introduces the scripts to generate the input data for the evaluation in Chapter 5. These python scripts generate *.csv* files and represent applicants applying for different seminars. We first look at how these *csv* generators work in general and then see how each script can be adjusted to get the desired input. Lastly, we introduce the different decision-making algorithms and the adaptions to the scripts.

4.1 Csv-Generator

This section covers the procedure of generating *csv* file representing the different applicants. (Each line represents one applicant.) With that, in each line, the information about this application is determined.

Parameters Each person has an attribute *time* representing the time at which the person applies for a seminar. Further, to uniquely identify each person, everyone gets a unique *id*. To execute fairness conditions in RTLola, we need the required parameters for each applicant. In our setup, every person has a gender, either male or female, which is determined by a coin flip. Also, each person applies for a seminar, which is also randomly determined based on the existing number of different seminars. To monitor counterfactual fairness (Section 2.1.3), each applicant has a *qualification* which is a random number between 1 and 10. A representing line for an applicant looks like this:

time	id	gender	seminar	qualification	decision	idDecision
1	1	F	1	8	#	#

Figure 4.1: Single Applicant

Note, that the two parameters *decision* and *idDecision* are for the decision for a person that might come later than the application. Since we want to investigate rather realistic

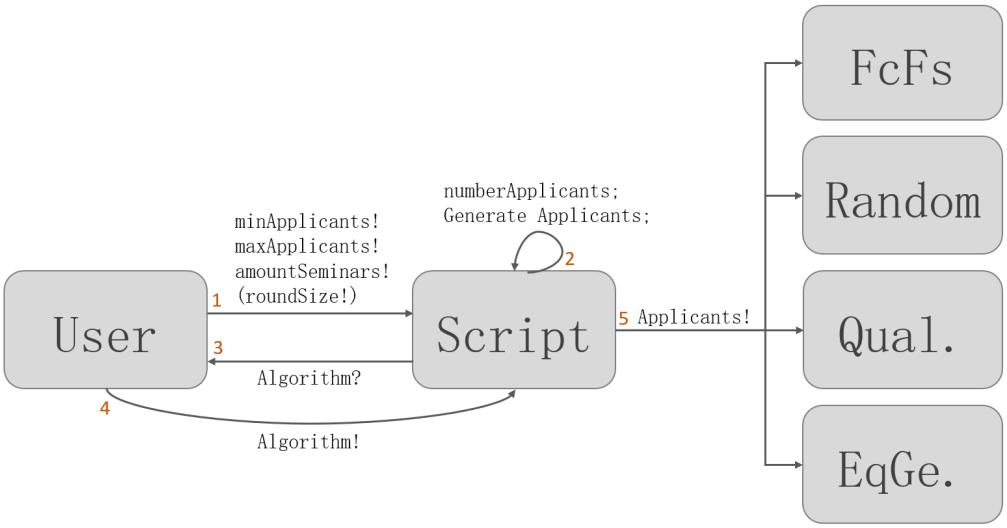


Figure 4.2: Control Flow for the Input Generation

inputs, there are two different output variants:

- The decision of whether a person is accepted or rejected is specified at a later point in time. A time delay is a random number from 1 to 100, representing the time "needed" to make a decision.
- There are bounded deadline "rounds", meaning that whenever a person applies, a threshold gets counted up by 1. If the threshold is reached, all persons that applied in this "round" get their decision at the same time and after that, the threshold gets reset. Further, in these tables, there is an extra parameter called *round*, which indicates in which iteration of rounds the process is currently.

For each of these two variants, there is a separate script, i.e. one script for the delayed version and one script for the specific "rounds". To realize the delayed decisions, each applicant has two extra parameters, as shown in Figure 4.1, one for the decision of being accepted, and one to specify to whom the decision belongs indicated by the *id*. As an example, if at any point in time, the parameter for the decision is true and the parameter for whom the decision counts is 4, the person with *id* of 4 gets accepted to the applied seminar. A later example visualizes this behavior.
The input generation procedure is represented in Figure 4.2. An exclamation mark ("!") indicates that these values are provided from the source where the edge originated. A question mark ("?") means the source "asks" the target for the specified value. Further, the numbers indicate the order of execution.

Input Generation The user of the script first specifies the minimum, as well as the maximum amount of applicants as shown in Figure 4.2. Then, the user specifies the number of different seminars, each with a capacity of 12. The script that features specific

"rounds" of applicants asks the user then to state the size of each round (In the figure in brackets). When there are as many applicants as specified in the size of "rounds", these applicants then get immediately their acceptance decision.
Then, the actual number of applicants gets calculated by taking a random number between the minimum and the maximum number of applicants. Now, each applicant gets generated with their information according to the description above and then gets appended to the current list of applicants. Finally, the user gets asked which algorithm should be used for decision purposes and according to that input, each applicant gets their decision of being accepted or rejected at a later point in time than the application of the person.
There exists also a third script, where steps three and four (Figure 4.2) do not exist and the script "sends" the applicants to every different algorithm. This process is done 100 times, i.e. step two, and then step five is executed 100 times (Figure 4.2). With that third script, it is possible to have different algorithms executed on the same input data. Hence, we can evaluate these datasets and get statistically accurate results.

4.2 Different Decision-Making Algorithms

In this section, we introduce the different decision-making algorithms supported by the scripts. We briefly explain the proceeding of each algorithm as well as the motivation to analyze them. In this thesis, we focus on four different decision-making algorithms:

First come, First served This algorithm accepts the first people applying regardless of any other metrics. In the input generation script, this is realized by checking for each applicant, if there is still capacity in the applied seminar, i.e. if this is the case, then the person gets accepted. With that, for each seminar people applied for, the first 12 get accepted, while all others get rejected.
We expect this algorithm to be unfair in most scenarios. However, this algorithm is still an interesting case, since there are cases in the real world where some sort of first come, first served is applied. Further, based on this algorithm we can see if the monitoring process in RTLola is executed correctly.

Randomize This algorithm randomizes the people that get accepted. This is done by picking randomly 12 people that applied for the seminar. Additionally, this procedure is repeated for each different seminar. All others then get rejected.
With that algorithm, we want to show that randomness generates a bias in general. With randomization, certain demographic groups get disadvantaged since randomization respects no specific parameter.
Concluding, we expect this algorithm to be unfair in most scenarios regarding (almost) all fairness conditions.

Best Qualification The approach of this algorithm is to accept the best 12 applicants for each seminar. I.e., the people with the highest qualification. We realize this in the input generation by checking the qualification of each applicant for each seminar and then accepting the best 12 for each seminar. All others then get rejected, because there is no capacity left in the seminars.
Since this algorithm aims to fulfill the condition from counterfactual fairness (Section 2.1.3) we expect this algorithm to be fair in most scenarios according to counterfactual

fairness. However, this might yield critical results, since the other fairness criteria may be violated. This would show that picking the people with the best qualification is not fair according to fairness definitions like demographic parity (Section 2.1.1) or conditional statistical parity (Section 2.1.2). This finding would be interesting since it is a common method in the real world to pick the best few applicants and with that, not respect other aspects like the equality of gender.

Equal Gender The equal gender algorithm ensures that the acceptance rates of males and females are (almost) the same. For that, the algorithm tracks the number of males, as well as the number of female applicants, and with that, calculates the acceptance rate and how many of each demographic group should be accepted.
By ensuring that the acceptance rates are roughly the same, we also expect demographic parity to never be violated towards the end of the scenario, since the desired acceptance rates should then be stabilized. We use this algorithm as a reference and to compare other forms of algorithms with it. Furthermore, this algorithm shows that ensuring that one fairness condition holds, does not mean that the algorithm is completely fair. Other fairness definitions may be violated frequently.

4.3 Generating Input Scenario

We now want to execute both input-generation scripts and inspect the corresponding outputs. The first script represents more of a scenario for a job application with no specific deadline. In contrast, the second script describes the seminar-assignment system that is mentioned in this thesis since it shows specific deadlines. For all people applying before that deadline, a decision is made immediately when the deadline is over.
We will now see an example for each version of the script:

4.3.1 Delay-Script

To obtain an input file, the script itself has to be executed. Then, it is necessary to specify the minimum and maximum amount of Applicants asked by the script, as well as the desired amount of seminars. When these parameters are specified, the script generates the input table and then asks the user to decide, which algorithm should be used in this scenario for the decision of acceptance for each applicant. The user then gets a list of available algorithms and specifies which one to use. After that, the script is done and yields the resulting input table.
We now generate a *.csv* file by running the delay script with the following attributes:
Let us say we want exactly 10 applicants, so we choose both the minimum, as well as the maximum of applicants to be 10. For 10 people it is sufficient to have 1 seminar, and we choose the algorithm to be "first come, first serve", which accepts the first people applying for a seminar.
The resulting *csv* file is shown in Figure 4.3
The table in Figure 4.3 presents a possible output generated with the delay script. In total 10 people applied for a seminar and all got accepted eventually to that seminar. Note that at times when no person applies for a seminar but a decision for another person is made, the first five parameters, except for the parameter *time*, just stay empty.
Further, the value of the parameter *decision* always refers to the applicant with the id of the value of *idDecision*. With that, e.g. at time five the "true" value in *decision* refers to the

time	id	gender	seminar	qualification	decision	idDecision
1	1	M	1	7	#	#
2	2	F	1	8	#	#
3	3	F	1	2	#	#
4	4	M	1	10	true	1
5	5	F	1	4	true	2
6	6	M	1	6	#	#
7	7	M	1	2	true	6
8	8	M	1	1	true	4
9	9	M	1	2	true	8
10	10	M	1	2	true	5
11	#	#	#	#	true	9
12	#	#	#	#	true	7
13	#	#	#	#	true	3
14	#	#	#	#	true	10

Figure 4.3: Input Table with Delay

person with the id "2". This concludes with the result that person with id 2 gets accepted to seminar 1 at time 5.
Also, no person got rejected from the seminar, since the capacity of the seminar is 12 and the chosen algorithm was "first come, first served".

4.3.2 Deadline-Script

Now we want to generate a csv file that features specific deadlines. For that, we execute the deadline script with again the number of applicants being exactly 10 and also with just 1 seminar. Further, we specify each "round" with 4, meaning that whenever 4 people apply, all of the m get their decision immediately after the 4th applied. We also choose "first come first serve".
The resulting csv file is represented in Figure 4.4. We see in this figure (Figure 4.4) that when the number of applicants in each round hits up to 4, all of them get a decision immediately. If there is a rest ($10\%4 = 2$) the rest also get their results immediately, even though the limit is not reached for the round. Since the limit will never be reached, these applicants also need a decision. Also the parameter *round* indicates in which iteration of rounds we are currently. This information is required to monitor these scenarios in RTLola. We also see that within the lines of the decision for the applicants, the time always stays the same, since we want every person from each round to get their result at the same time as each other.

In conclusion, we introduced the proceeding of generating input in the form of *csv*-files in this section. Further, we briefly explained the different decision-making algorithms that are investigated in this thesis.
With these algorithms being introduced, we can now complete our *csv*-files by adding the decisions made in the specified algorithm. Now, we can run these algorithms through experiments. In the following section (Section 5) we analyze the behavior of these algorithms in generated-practical scenarios.

time	id	gender	seminar	qualification	decision	idDecision	round
1	1	M	1	8	#	#	1
2	2	F	1	5	#	#	1
3	3	F	1	6	#	#	1
4	4	F	1	1	#	#	1
4	#	#	#	#	true	1	1
4	#	#	#	#	true	2	1
4	#	#	#	#	true	3	1
4	#	#	#	#	true	4	1
5	5	M	1	2	#	#	2
6	6	M	1	9	#	#	2
7	7	M	1	10	#	#	2
8	8	F	1	9	#	#	2
8	#	#	#	#	true	5	2
8	#	#	#	#	true	6	2
8	#	#	#	#	true	7	2
8	#	#	#	#	true	8	2
9	9	M	1	2	#	#	3
10	10	F	1	5	#	#	3
10	#	#	#	#	true	9	3
10	#	#	#	#	true	10	3

Figure 4.4: Input Table with Rounds

Chapter 5
Experiments

In this section, we evaluate the different algorithms from Section 4.2 with the specifications of Section 2.1.3 regarding fairness. We analyze the frequency of occurring triggers, as well as the timing of these triggers, i.e., when these triggers occur within the scenarios. Further, we check how long the state stays unfair. We then interpret these results to see the suitability of each algorithm. Further, we introduce an example of "Simpson's Paradox" (Section 2.1.3) and show that we can detect occurrences of that with the specifications from Section 2.1.3.

5.1 Trigger Behavior in Demographic Parity

In this section, we analyze the different decision-making algorithms introduced in Section 4.2, namely "Equal Gender", "First come First served", "Best Qualification" and "Randomized" with respect to demographic parity (Definition 2.1.1). We compare each algorithm regarding the total amount of triggers, as well as the temporal timing of triggers. We then interpret these results according to the rank of fairness.
For the following experiments, 1000 input files were generated with the random delay approach (Section 4.1), i.e. the decision for each applicant is made with a delay of 1-100 discrete points in time. Then, each algorithm made decisions for these input files. The result is 1000 files for each different algorithm, i.e. 4000 in total. Further, each input file has precisely 100 applicants and there exist three different seminars, resulting in 36 spaces in total for each input file.

5.1.1 Frequency of Triggers

The specified input tables run against the implementation of demographic parity from Section 3.4.1. Hereby, it is important to set the confidence κ (Definition 3.3.1), as well as the tolerance ϵ for the trigger to a matching value. Therefore, in Figure 5.1 different confidences κ with different tolerances ϵ are compared.

This bar chart shows the number of thrown triggers for each algorithm.

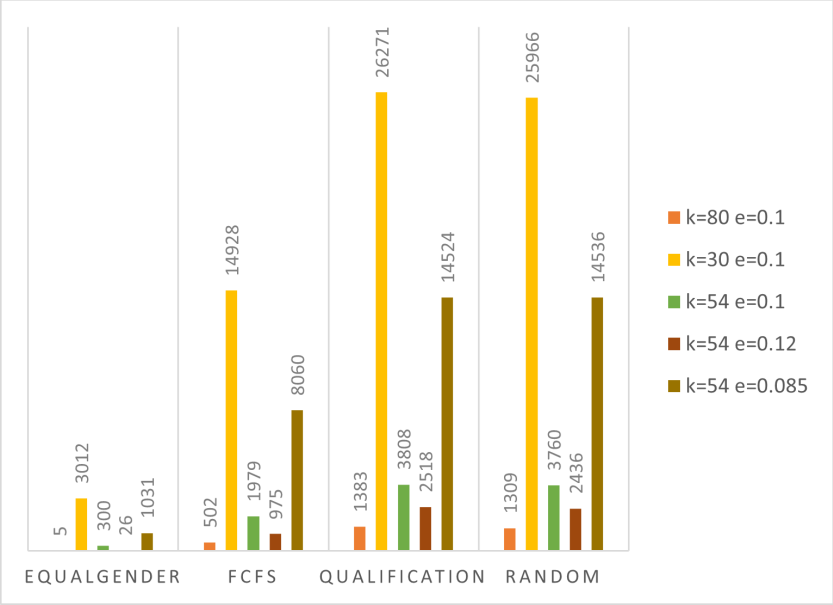


Figure 5.1: Total Amount of Thrown Triggers

By setting the confidence κ to 80, the amount of thrown triggers does not yield accurate results. Even though there are only five triggers raised by the "Equal Gender" algorithm, which would be an argument supporting the setting, this specific value for κ is not suitable. The other algorithms also raise subnormal numbers of triggers, e.g., "Randomization" only raised 1309 triggers in total, meaning that 1383 decisions out of 100.000 decisions are unfair according to this algorithm. Since the expected behavior of this algorithm differs from these results, this setting does not match real-world scenarios.
By assigning a value of 30 to the confidence κ, which is too low, there are too many triggers thrown to be accurate. The "Equal Gender" algorithm raises 3012 triggers, though, since its expected behavior is to make decisions fairly, this value for κ is not suitable as well.
If the confidence κ is set to 54 and the tolerance ϵ to 0.12, the results might seem to be realistic. However, in that case, "FcFs" only raise 975 triggers, which is too little for a seemingly unfair algorithm. Therefore, this setting is not used in further experiments as well.
When setting the tolerance ϵ to 0.085 while keeping the confidence κ at 54, we see that there are thrown too many triggers to match realistic results. Since "Equal Gender" tries to ensure the equality of acceptance rates between the demographic groups, 1031 thrown triggers indicate a too strict tolerance.
Resulting from that, the setting where κ is set to 54 and ϵ to 0.1 serves the most realistic results. While "Equal Gender" throws a feasible amount of triggers, the other algorithms seem to face some issues regarding fairness, which is expected since within their calculations, these algorithms do not regard acceptance rates of males and females.
Concluding, the following experiments, i.e. the timing of triggers, as well as the period of unfair states are executed with κ being 54 and ϵ being 0.1. We now analyze the result of this experiment regarding this specific setting:

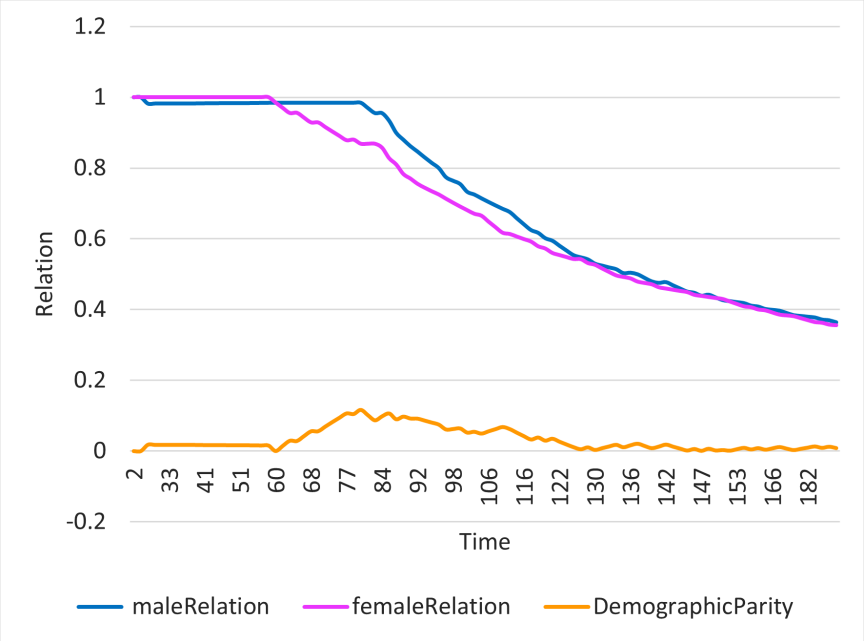


Figure 5.2: Acceptance Rates over Time with Equal Gender

Equal Gender Iterating over 1000 input files yields 300 triggers in total, which is by far the lowest number of all algorithms. This is an expected behavior since the equal gender algorithm tries to balance the acceptance rates of both demographic groups to decrease the margin. Since demographic parity checks the discrepancy between the acceptance rates of both demographic groups, the condition mostly holds and therefore the number of triggers is proportionally little.
However, there are still some scenarios that are unfair according to demographic parity. The reason for that may be a local unequal distribution of demographic groups. Though this might happen in real-world scenarios, one cannot ensure that different demographic groups apply in alternating order. Also, this is the reason why the "Equal Gender" algorithm still has triggers, and further, the algorithm only ensures gender equality at the end of a scenario, meaning that there can occur local inequalities between both demographic groups. An unequal distribution of different demographic groups is shown in Figure 5.2.
Especially between time 77 and 90, the discrepancy between both acceptance rates grows. This is caused by the fact that many females get rejected in this period, while there are (almost) no decisions for males. This leads to a bigger margin between these acceptance rates and results in thrown triggers. However, it is also worth noting that after that time period, the discrepancy gets smaller, and acceptance rates have almost equal values (~ 0.0084 discrepancy) by the end of the table. We analyze this special behavior in later experiments.
In conclusion, we decide "Equal Gender" is a fair algorithm in general, even though there are thrown triggers. Since these thrown triggers result from an unequal distribution between demographic groups, and ultimately the decisions in an input file get fair, this algorithm can be seen as "fair".

First Come First Served In Figure 5.1, "First come First served" (FcFs) has 1979 triggers thrown in total. This is far worse than "Equal Gender" with ~ 6.5 times as many triggers are thrown. With ~ 2 triggers thrown on average per input file, this algorithm cannot be seen as fair in general. However, "First come First served" performs notably better than "Best Qualification" and "Randomized". This is caused by the natural property of FcFs to accept all applicants in the beginning while rejecting all excessive applicants at the end of the input table.
Choosing FcFs as the algorithm, the first few applicants always get accepted (at least the first 12 applicants, since there is always as least one seminar). In this stage, no unfairness according to demographic parity (Definition 2.1.1) can occur since the acceptance rates of both demographic groups stay a 1.0. Then, there is a stage where mixed decisions occur. The reason for that is that some seminars are already booked full, while some of them still have some vacancies. Within that stage, the acceptance rates can change rapidly, dependent on the distribution of males and females. In the final stage, all remaining applicants get rejected, since all of the available slots are already occupied. Also, in this stage, the discrepancy between both acceptance rates does not change significantly. Thus, this also includes that if at the end of stage two, the acceptance rates are within the tolerated bound, this will not change anymore and therefore little triggers are thrown.

Best Qualification Iterating over 1000 input files with this algorithm, demographic parity recognized 3808 unfair positions. Since "Best Qualification" picks the applicants with the highest qualification, it does not regard gender. Hence, the distribution of accepted males and females is not respected in the calculations of the algorithm, and therefore the acceptance rates can differ significantly. Thus this may result in unfair scenarios concerning gender.
All in all Figure 5.1 shows that "Best Qualification" is not a suitable algorithm when taking respect to demographic parity. Picking the "best" applicants does not ensure gender equality and therefore, this algorithm does not behave fair according to demographic parity.

Randomization As shown in Figure 5.1, this algorithm behaves equally unfairly as "Best Qualification". However, this is expected behavior since choosing the accepted applicants randomly does not ensure gender equality. Thus, since this is the critical criterion of demographic parity (Definition 2.1.1) "Randomization" behaves unfairly according to this fairness definition.
In conclusion, this specific algorithm is not appropriate when trying to ensure fairness. Even with an equal distribution of demographic groups, this algorithm can behave unfairly and therefore is not applicable in real-world scenarios.
Though, this algorithm can be seen as a reference to whether the implemented specifications (Section 3.4.1,3.4.2,3.4.3) work properly. With that, the expected result is an unfair behavior of the "Randomized" algorithm.

We now analyzed the accuracy of algorithms regarding the total number of thrown triggers. With that, "Equal Gender" performed the best, making it a fair algorithm according to demographic parity. All others were significantly worse and contained a lack of fairness by not respecting the acceptance rates between both demographic groups.
Further, we introduce temporal aspects of the experiments in the next section. We compare the timing of triggers in the next section and interpret these results.

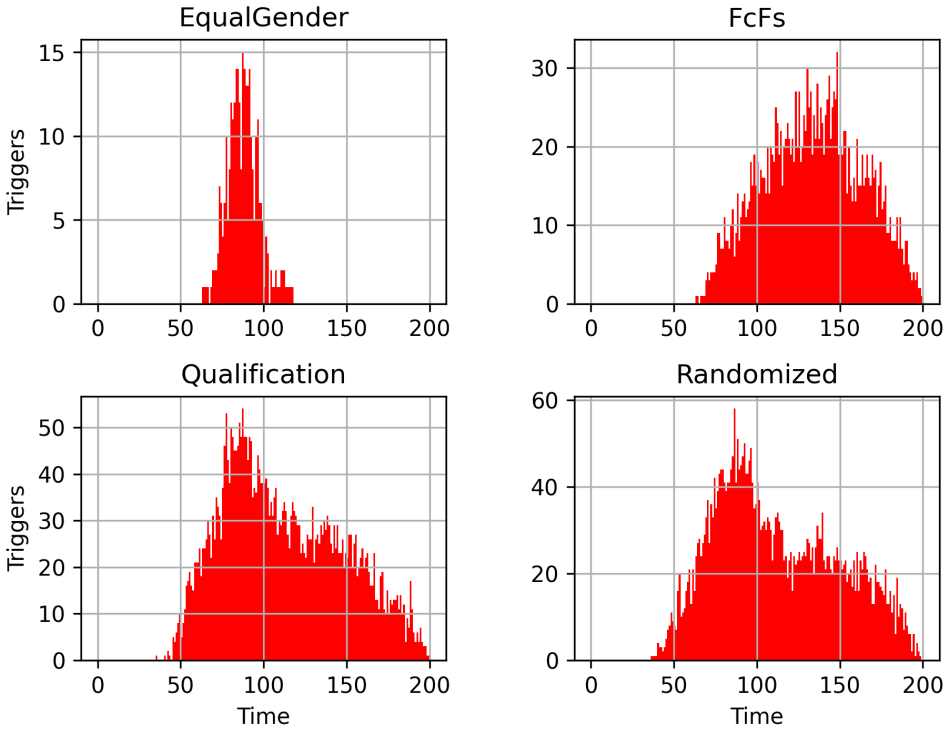


Figure 5.3: Temporal Distribution of Triggers

5.1.2 Timing of Thrown Triggers

We now extract the timing of thrown triggers and according to that, analyze the behavior of each algorithm according to demographic parity. The histograms in Figure 5.3 show the temporal distribution of thrown triggers. In all histograms, there are no triggers thrown in the early stage of the input files. This is caused by the application of MAP and Normalization (Definition 3.3.1). With that, no triggers are thrown in the beginning since it is not possible to tell whether an algorithm behaves fair after only a few decisions.

Equal Gender With "Equal Gender", most triggers are thrown in the time between 70-90. The impact of applying MAP and Normalization to the acceptance rates yields this behavior. Further, after time ~ 90 the number of thrown triggers goes rapidly down. The latest trigger thrown is at time ~ 120. This again supports the recognition from the earlier experiment (Figure 5.1) that "Equal Gender" gets fair ultimately. Further, this histogram (Figure 5.3) shows that unfairness according to demographic parity when using equal gender can only occur in the middle of input tables. The first decisions cannot get unfair because of MAP, and the last decisions are not unfair because of the behavior of the algorithm. Only in the time between the impact of MAP and the stabilization of the acceptance rates, there can occur local unfairness.
Concluding, this histogram supports the statement from the experiment earlier (Section 5.1.1), namely that this algorithm is suitable regarding demographic parity. It may behave unfairly locally in specific scenarios, but it ensures to be fair ultimately.

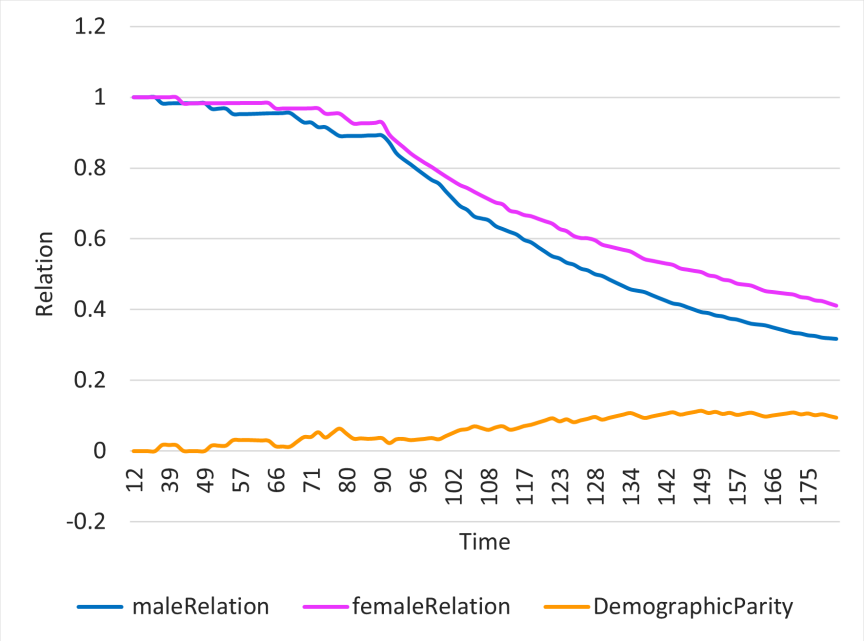


Figure 5.4: Acceptance Rates over Time

First Come First Served From the histogram, we can see, that the timing of thrown triggers has similarity to the normal distribution. The peak is at time $\sim 140 - 150$, which is later than with "Equal Gender". After that, the number of triggers decreases up to time 200 due to the reason that the decision for a person is made with a delay of 1-100 timestamps. Resulting from that, the length of each input file differs and therefore, there are fewer decisions made at later points in time. With that, the total number of triggers decreases over time. Though, this behavior does not support the statement that the algorithm gets fairer at later time points.
The histogram for "First come First served" therefore strengthens its ineffectiveness regarding demographic parity, since one can see that if there is a trigger at a certain point in time, the trigger usually keeps up until the end of the input file. This behavior can be seen in Figure 5.4.

The lines representing the acceptance rates of each demographic group in Figure 5.4 show that until time ~ 90 the acceptance rates are almost stable at 1.0-0.9. After that, there is a downward trend, since now (almost) all slots from the seminars are occupied. Also note, that FcFs accept the first applicants, but this does not imply that the first decisions are positive. From the time a decision is made for a specific person, negative decisions may occur before positives (e.g. the first few applicants all get a high delay while the first applicant, where a seminar is booked full, gets a delay of 1). In this situation, there is some impact at the beginning that lowers the acceptance rate of the according demographic group.
Further, it is noticeable that the discrepancy between the acceptance rates of both demographic groups grows over time, especially from time ~ 100 until the end of the input file. This behavior is reflected in Figure 5.3.

Best Qualification The histogram for "Best Qualification" in Figure 5.3 visualizes a high amount of thrown triggers at time $\sim 70 - 80$. After that, there is a constant downward trend up to time 200. Further, the number of thrown triggers is higher than with "Equal Gender" or "FcFs" (Figure 5.1).
This result can be interpreted as the unsuitability of this algorithm according to demographic parity. After the impact of MAP, the number of thrown triggers goes rapidly up. After that, the total number of triggers reduces per time, though ultimately not getting fair.

Randomization For "Randomization", the histogram from Figure 5.3 shows a similar effect as for "Best Qualification". Early, after the impact of MAP, the number of triggers rises to a peak of almost 60 triggers at a specific time. After that, there is a downward trend, which is caused by the different lengths of each input file. Since some files are shorter than others, the total number of triggers decreases over time.
However, at the end of the input files, there are still errors thrown, leading to the fact that "Randomization" is also not suitable, since the algorithm does not respect the acceptance rates in its calculations.

In this section, we investigated the temporal timing of occurring triggers. With that, it was possible to interpret the rank of fairness for each different algorithm.
In the next section, we analyze the rising edges.

5.1.3 Occurences of Rising Edges

We now investigate the occurrences of rising edges. Rising edges are points in time, where currently a trigger is thrown but the last time the acceptance rates were checked, there is no trigger thrown. With that, it is possible to observe, whether an algorithm is close to the tolerated fairness bound. When there are many rising edges relative to the total number of triggers, this means that the trigger is often thrown for a short period. In contrast to that, if there are few rising edges relative to many total triggers, this means that when a trigger is thrown, this trigger remains for a long period of time and therefore the algorithm is behaving unfairly according to demographic parity.
Figure 5.5 shows the ratio of rising edges.

In this figure (Figure 5.5), the orange bars represent the total amount of thrown triggers (as in Figure 5.1) while the blue bars show the number of rising edges. For "Equal Gender", almost $\frac{1}{3}$ of each trigger is a rising edge. This further supports the statement that unfairness in this algorithm is only local since otherwise, the ratio of rising edges to the total number of triggers would be less.
Regarding all other algorithms, the ratio lies between 0.164 and 0.176, which is significantly less than for "Equal Gender". With that, all other triggers, that are not rising edge, have to follow up directly from a previous trigger. This results in a trend where the period of triggers is longer and therefore the algorithms stay longer unfair and/or stay unfair until the end of the input table. This behavior was already shown in earlier experiments, however, with Figure 5.5, this statement gets strengthened further.

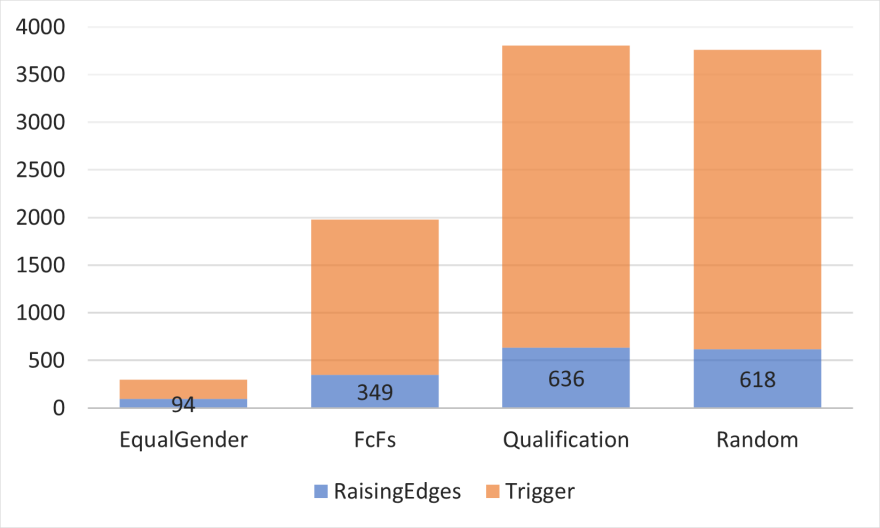


Figure 5.5: Rates of Rising Edges

5.1.4 Trigger Behavior in Bounded Deadline Rounds

This section covers an analysis of the trigger behavior using the approach of having specific deadlines within the input files (Approach 4.1). Every time after 20 new applicants, each of them gets a decision, all at one later point in time than the 20^{th} applicant. Besides this difference, the framework is the same as in earlier experiments, i.e. 1000 input files for each algorithm with 100 applicants within each input file. In each scenario, there are three seminars with a capacity of 12. Further, in demographic parity a confidence κ (Definition 3.3.1) of 54 and a bound of 0.1 is chosen.
With this setting, Figure 5.6 shows a bar chart representing the total number of thrown triggers for each seminar.

Figure 5.6 shows a seemingly similar chart as with the random delay approach. However, in "Equal Gender" there are almost no triggers thrown with less than $\frac{1}{3}$ thrown triggers. Also, in all other different algorithms, there are fewer triggers thrown, though the relative difference to the random delay script is less with only $\sim$ 14.4%-18% fewer triggers thrown. Thereby, the reason for this significantly better performance of "Equal Gender" is that the local unequal distributions regarding the demographic groups do not harm the outcome of that algorithm that much anymore. This is because there are always made 20 decisions at once, so, in general, the local inequality can be balanced again.
For the other algorithms, since these do not regard the acceptance rates in their calculations, this behavior only gives slightly better performance.

Summarizing, regarding demographic parity, "Equal Gender" is a suitable algorithm while all others lack fairness. Though we also saw, that even an algorithm that tries to keep the acceptance rates the same may come up with some local unfair sections. Further, demographic parity is a feasible fairness condition that can be computed in many different scenarios. However, some experiments showed that one cannot rely on single-thrown triggers, since local unequal distributions can disturb the acceptance rates for a certain period of time.

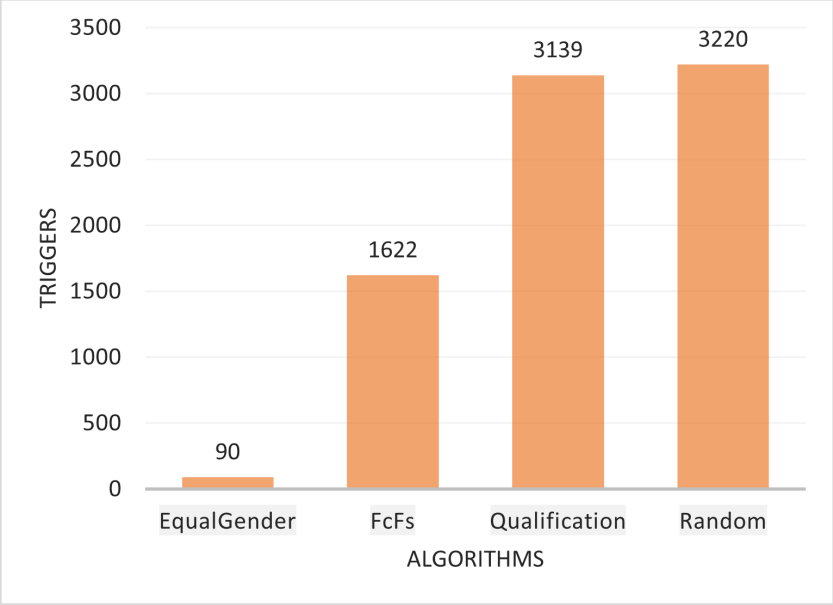


Figure 5.6: Total Amount of Thrown Triggers

The next section covers experiments for statistical conditional parity.

5.2 Trigger Behavior in Statistical Conditional Parity

This section covers experiments regarding conditional statistical parity (Definition 2.1.2), as well as the analysis and interpretation of results. For that, the structure of the experiments is similar to the experiments for investigating demographic parity (Section 5.1).

5.2.1 Frequency of Triggers

We now use the implementation of conditional statistical parity from Section 3.4.2 to investigate the generated input files for unfairness. We thereby use the same approach as for the experiments for demographic parity in Section 5.1.1 to find suitable values for κ and ϵ. We then analyze and interpret the result for the different decision-making algorithms. The bar chart in Figure 5.7 shows the total amount of thrown triggers for each algorithm.
Further, we introduce a modified version of the "Equal Gender" algorithm. Since the original version only ensures the equality of the acceptance rates between both demographic at the end of the input table, this algorithm is not feasible as a reference for conditional statistical parity. Though we want at least one algorithm to serve as a reference, we adjust the current version of the "Equal Gender" algorithm. Thereby, the modified version ensures, that the acceptance rates are equal within each seminar. To do that, the algorithm calculates the number of males and females that should be accepted to each seminar to have (almost) equal acceptance rates as a result. We then can use this new version of the algorithm as a reference.

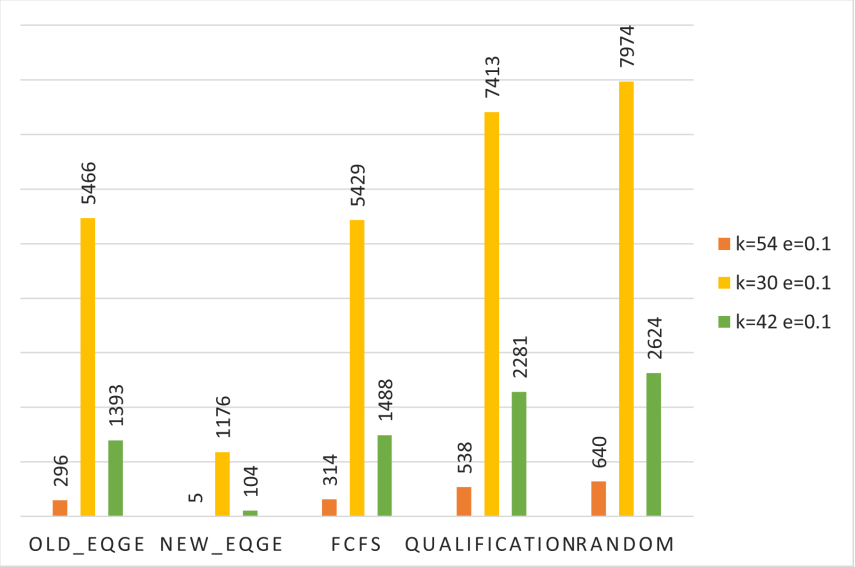


Figure 5.7: Triggers thrown with Conditional Statistical Parity

Figure 5.7 shows the incompatibility of the original version of the "Equal Gender" (*OLD_EQGE*) algorithm regarding positive reference. The reason for that is, it behaves almost in the same way as "First come First served" regarding the amount of thrown triggers. Therefore, we will not cover further experiments with this algorithm. Instead, the experiments are executed with the new adjusted version of this algorithm.
Further, taking a look at the different confidences, a too-high value, i.e. $\kappa = 54$ results in little triggers raised in general. Though, it is unlikely that the "Randomized" algorithm behaves fair according to conditional statistical parity (Definition 2.1.2), a lower value of κ is required. Also, taking the value 30 as confidence, the result is not suitable, since even with the adjusted version of the "Equal Gender" algorithm, there are still over 1000 triggers thrown. Therefore, a confidence of 42 is realistic since there are few triggers thrown with the seemingly most fair algorithm, but algorithms that do not take the acceptance rates into account are raised with significantly more triggers.
Now, we analyze the number of triggers regarding the confidence κ of 42:

While there are only 104 triggers thrown using the modified "Equal Gender" algorithm, "FcFs" raised over 1400 triggers and both other algorithms contain more than 2000 triggers. We therefore can determine the "Equal Gender" algorithm to be fair regarding conditional statistical parity, even though there occur triggers. The occurrence of these triggers is based on local unequal distributions of demographic groups, as well as the fact that the algorithm ensures equal acceptance rates at the end of an input file though, not at every timestamp.
This behavior can be seen in Figure 5.8, where we investigate the behavior of the algorithm in a specific scenario.

Equal Gender Within seminar one, there is a discrepancy between the acceptance rates of both demographic groups at times 51-76, though, after that the margin between the acceptance rates gets smaller and eventually stabilizes. Further, we see, that at the end of

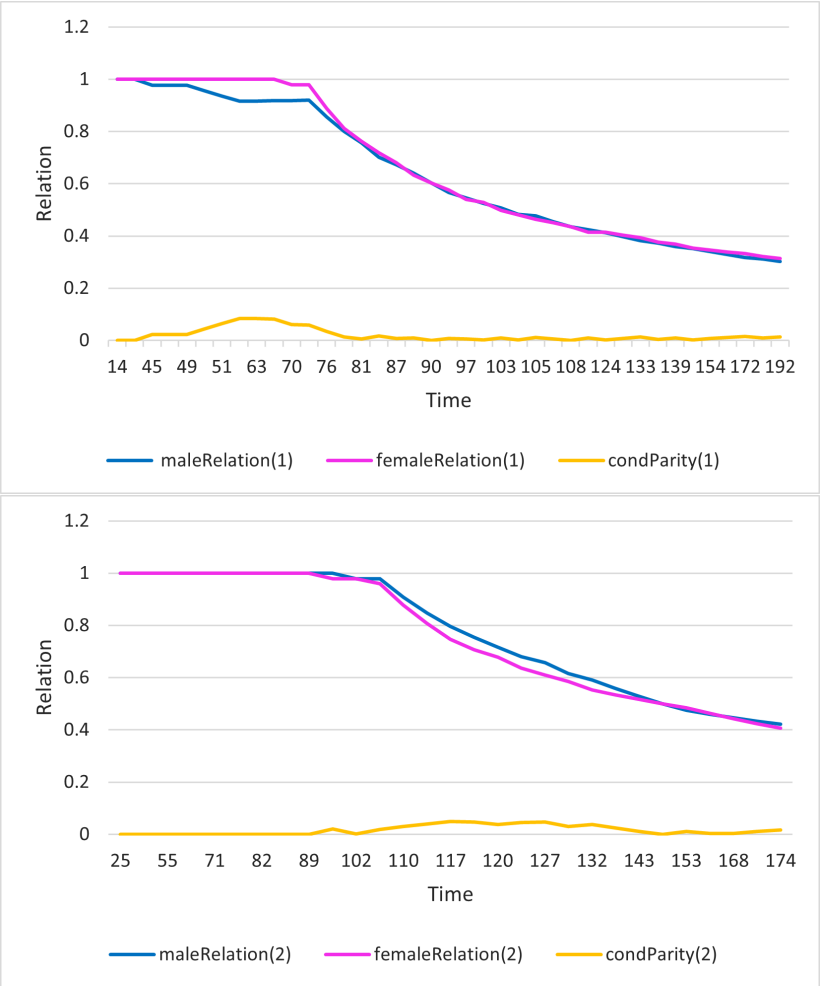


Figure 5.8: Acceptance Rates over Time with Equal Gender

the scenario, the acceptance rates are almost the same which is caused by the behavior of the calculations of the "Equal Gender" algorithm.
For seminar two, there is almost no difference between the acceptance rates. This is common when using this algorithm since if there is an equal distribution of demographic groups, the algorithm decides to keep the acceptance rates the same. When there is no unequal distribution, there does not occur local unfairness. Again, within this seminar, we see that the algorithm ensures (almost) equal acceptance rates at the end of the scenario.
In the third seminar, there occur some triggers. Between times 74 and 79, the difference in the acceptance rates (~ 0.113) is higher than the tolerance ($\epsilon = 0.1$). Figure 5.8 shows the acceptance rates of males decreasing since time 53 while until time 75 every female got accepted. This results in local unfairness. Though, ultimately the acceptance rates stabilize and at the end of the input file, the acceptance rates are (almost) the same. Further, there are only two triggers raised in total, even though the discrepancy of the acceptance rates in seminar three is higher than 0.1 at times 74-79. The reason for that is in the time between 74 and 79 there are computations done for other seminars, and these

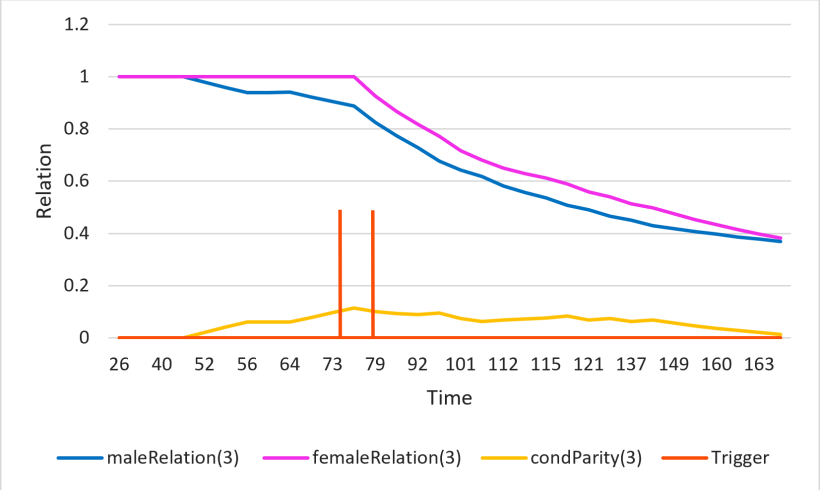


Figure 5.9: Acceptance Rates over Time with Equal Gender

do not show unfairness which results in no trigger thrown for these timings.
To summarize, the "Equal Gender" algorithm is suitable regarding conditional statistical parity, even though there occur some flaws. Since the algorithm only ensures equality of acceptance rates at the end of each seminar, it does not respect the acceptance rates at specific timestamps. With the usage of MAP and normalization 3.3.1, there cannot occur triggers at the beginning because single decisions do not weigh enough to decrease the acceptance rate to a critical value. Further, at the end of the input table, the algorithm stabilizes the acceptance rates in its computations, leading to the fact that unfairness can only occur in the middle of a scenario, and therefore is only a local unfairness.

First Come First Served According to Figure 5.7 there are 1488 triggers raised in the setting of $\kappa = 42$ and $\epsilon = 0.1$. Compared to "Qualification" or "Randomized", this algorithm performs between 1.5 and 1.76 times better than these regarding triggers. Even though this algorithm does not regard acceptance rates in its calculations, the behavior of this algorithm delivers some advantages. Since always the first applicants get accepted, there cannot occur unfairness, since the acceptance rates of both demographic groups are at 1 in the early stage of a scenario. Further, towards the end of a scenario, every excessive applicant gets rejected which results in both acceptance rates not changing significantly anymore. If then the discrepancy between the acceptance rates is within the tolerance bound, towards the end of an input table, there are no triggers raised, since the acceptance rates do not change decisively anymore. Though, if this is not the case, there are thrown many triggers because of this behavior. With that, "First Come First Served" can not be considered a fair algorithm regarding demographic parity because of its ignorance of gender equality in its calculations.

Best Qualification This algorithm performs unfairly according to demographic parity with 2281 raised triggers. Since with the algorithm, the only metric that is regarded in its calculations is the qualification of an applicant, it does not regard the demographic group the person belongs to. With that, unfairness regarding gender equality occurs, and therefore, demographic parity raises triggers.

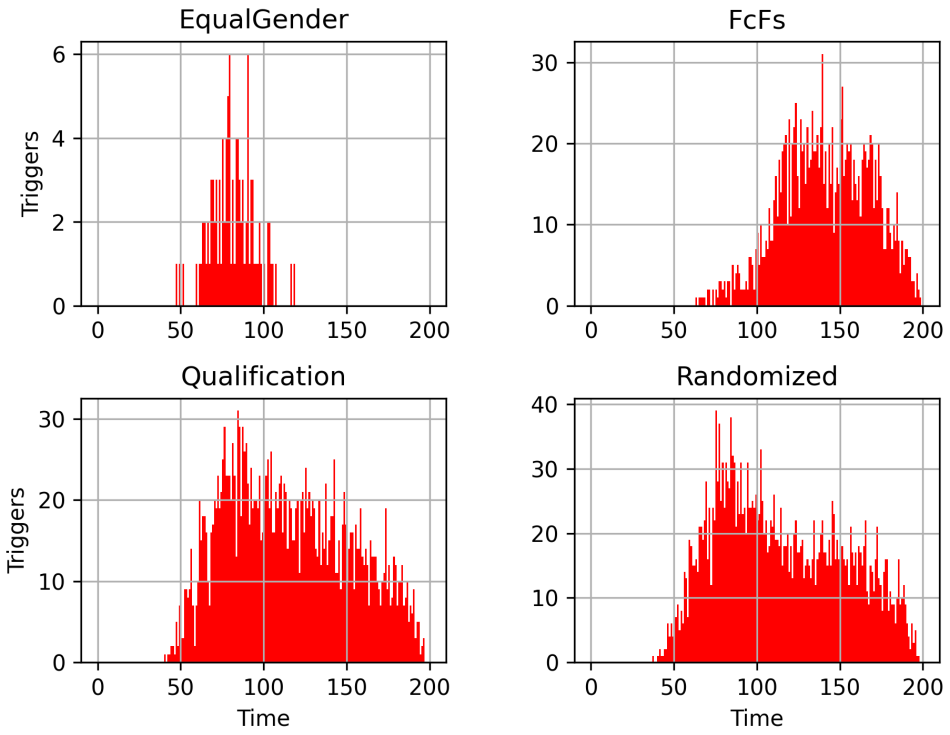


Figure 5.10: Temporal Distribution of Triggers

Randomized Similarly to "Best Qualification" and "First come First served", this algorithm does not decide to stabilize the equality of acceptance rates. Since this algorithm randomly chooses 12 applicants from each seminar, there occurs unfairness regarding gender equality, leading to thrown triggers by demographic parity, since the discrepancy of triggers is too high to be considered fair.

We now investigate the frequency of triggers with suitable confidence and tolerance. The next section covers the analysis of the temporal timing of raised triggers.

5.2.2 Timing of Thrown Triggers

In this section, we analyze and interpret the timing of raised triggers within each decision-making algorithm. Figure 5.10 shows the temporal distribution of raised triggers. Hereby, also 1000 input files with 100 applicants and three seminars each are investigated. These files are then monitored with the implementation of conditional statistical parity (Section 3.4.2) using $\kappa = 42$ as confidence and $\epsilon = 0.1$ as tolerance.

Further, we see the impact of MAP and normalization, since up to time ~ 45 there are no triggers thrown in any algorithm. This behavior is desired, since we do not want to make statements about the fairness of an algorithm while only regarding a few decisions. Also we can see in the histogram of "First come First served" that the first triggers are thrown late relative to the other algorithms. This supports the statement from earlier experiments, namely that because of the behavior of this algorithm, the discrepancy of the acceptance rates cannot differ significantly in the early stage of a scenario since all

applicants get accepted. However, we also see the delay of incoming triggers within that algorithm. Also, the reason for the decreasing amount of triggers per time towards timestamp 200 is the different lengths of each input file. Since the delay of a decision is between 1 and 100, some input files do not reach up to time 200.
Regarding "Equal Gender", the behavior in Figure 5.10 is the same as in Figure 5.3. Because of the behavior of the algorithm to be fair towards the end of a scenario, there is no insurance for gender equality during the scenario. Therefore, some local unfairness can occur. Note, that the dimension of the histogram for "Equal Gender" in Figure 5.10 is smaller than for all other algorithms. There are at most six triggers thrown in total for a single point in time, while in all other algorithms, there exists a point in time where over 30 triggers were raised. However, there are further no triggers raised after time ~ 120 which underlies the argument of the algorithm to be fair eventually.
Both "Best Qualification" and "Randomized" seem to throw triggers with the decreasing impact of MAP and normalization. Further, the number of triggers decreases over time towards the end of an input file, though this is the same reason as for "First come First served". There is no insurance that the acceptance rates stabilize to an (almost) equal value. Both algorithms can be interpreted as unfair towards conditional statistical fairness.

The results of this experiment lead to the conclusion that the adapted version of the "Equal Gender" algorithm is suitable regarding conditional statistical parity while monitoring the other algorithms seems to yield unfairness.
To strengthen this argument, the next section investigates the occurrence of rising edges.

5.2.3 Occurences of Rising Edges

This section covers the proportion of rising edges. Rising edges are points in time, where currently a trigger is raised, i.e. the discrepancy is larger than 0.1, but the previous iteration of these computations was within the tolerance bound.
When there is a high number of rising edges relative to the total number of thrown triggers, this means that the discrepancy between acceptance rates of demographic groups is close to the tolerated value. Also, many rising edges mean the discrepancy to get lower than 0.1 again in order to have the next rising edge.
Figure 5.11 shows the proportion of rising edges.

Here, the orange bars show the total amount of triggers thrown for each algorithm as in Figure 5.7, while the blue bars represent the number of rising edges. For "Equal Gender" there are 64 rising edges out of 104 triggers which is ≈ 0.615 as proportion. Further, we recognize that for all other algorithms, the ratio is similar. With that, the ratio for "FcFs" is $\frac{976}{1488} \approx 0.66$, for "Best Qualification" it is $\frac{1456}{2281} \approx 0.64$ and with the "Randomized" algorithm the ratio of rising edges to the total number of thrown triggers is $\frac{1670}{2624} \approx 0.64$.
This result differs from the experiment for demographic parity (Section 5.1.3), where the ratio of rising edges using "Equal Gender" is significantly higher than using other algorithms.
Since with conditional statistical parity, computations are executed simultaneously for each seminar, rising edges occur more likely. The reason for that is if there occur little rising edges, i.e. the thrown triggers are continuous, multiple conditions have to be fulfilled, which is more unlikely than with demographic parity. To get continuous triggers, every computation of the discrepancy between both demographic groups has to be greater than 0.1 within a specific time frame. Since these computations are done simultaneously, either every seminar has to show unfairness in this time frame, or there

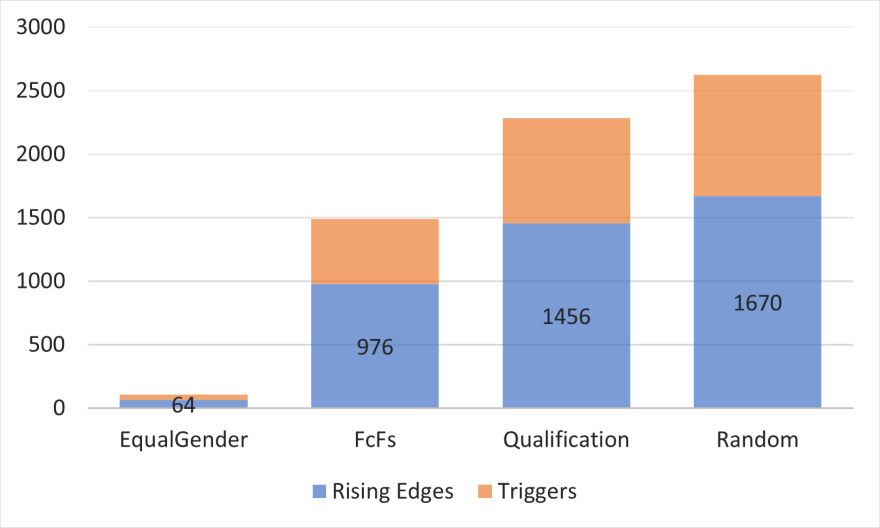


Figure 5.11: Rising Edges

are only computations done for seminars that are currently unfair. Since the timing of a decision, as well as the seminar the person is applying for is random-based, it is unlikely that these conditions are fulfilled. Therefore, the overall ratio of rising edges is higher than with demographic parity (Section 5.1.3).

In this section we analyzed and compared the decision-making algorithms from Section 4.2 with the implementation of conditional statistical parity (Section 3.4.2).
The following section covers the possibility of detecting Simpson's Paradox using the specifications for fairness definitions proposed in this thesis.

5.3 Detecting the Simpson's Paradox

In this section, we show the procedure for detecting an occurrence of Simpson's Paradox 2.1.3. For that, a specific scenario is investigated according to demographic parity and conditional statistical parity to find unfairness.
The upper Figure 5.12 shows a scenario with "Equal Gender" being the decision-making algorithm investigated for unfairness according to demographic parity. The upper line chart in Figure 5.12 shows unfairness between times 93 and 107. Since the acceptance rates differ more than 0.1 at that time, triggers are thrown. However, investigating the same input file regarding conditional statistical parity, there cannot be found unfairness at all. The lower line chart in Figure 5.12 shows the discrepancy of acceptance rates within each seminar. At every point in time, in every seminar, the difference in acceptance rates is smaller than 0.1 which leads to a fair scenario according to conditional statistical parity. With that, this scenario is exposed as Simpson's paradox (Section 2.1.3) since the outcome of the investigation for the critical group differs from the result of the investigation within each subgroup.
This scenario of an application process showed that it is indeed possible to detect occurrences of Simpson's paradox using the implementation of the fairness definitions introduced in this thesis.

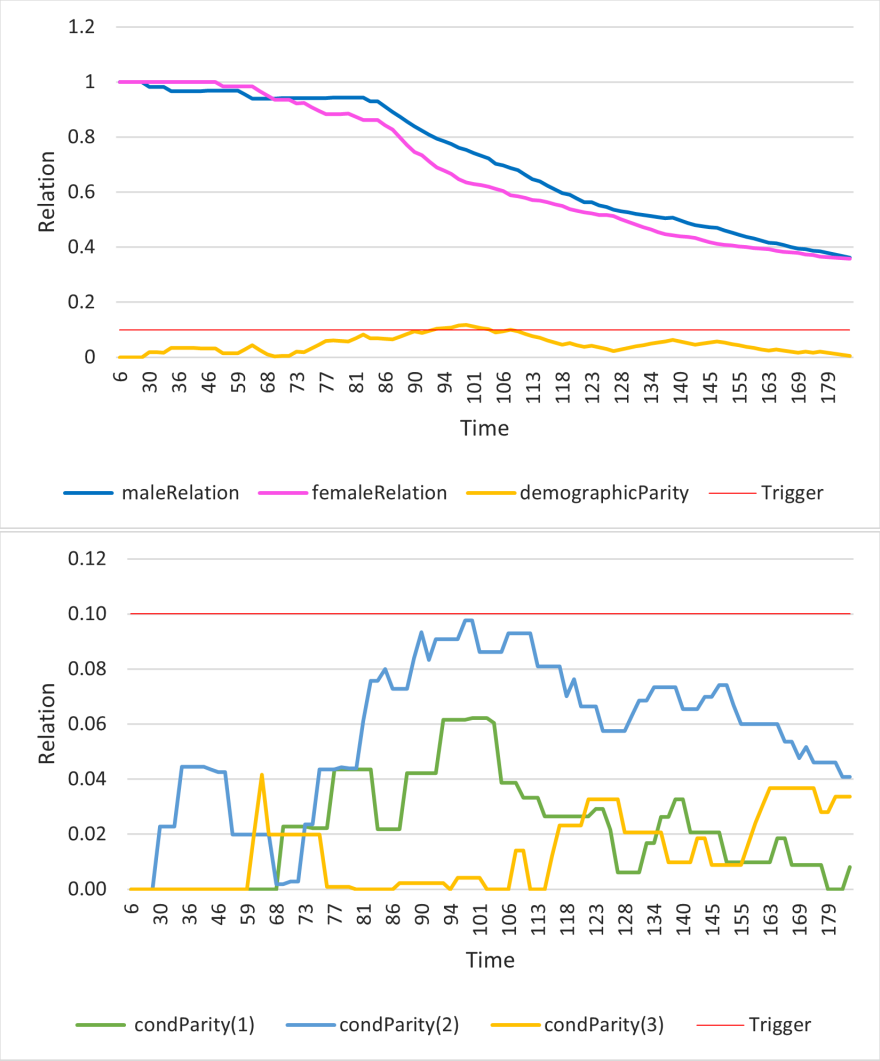


Figure 5.12: Acceptance Rates with Equal Gender

The next section checks the suitability of counterfactual fairness (Section 2.1.3).

5.4 Suitability of Counterfactual Fairness

In this section, we check whether counterfactual fairness (Section 2.1.3) is applicable to realistic examples. We, therefore, run through the same experiment for the frequency of triggers as for the other fairness definitions.
Figure 5.13 represents the number of raised triggers for each algorithm when the decisions are monitored using the specification for counterfactual fairness (Section 3.4.3). Since there are 100.000 decisions made from each algorithm, over half of them are unfair according to counterfactual fairness when using the "Randomized" algorithm. Choosing the accepted applicants randomly does not ensure that applicants with the same qualification get the same result. Also, if two applicants with the same qualification and the

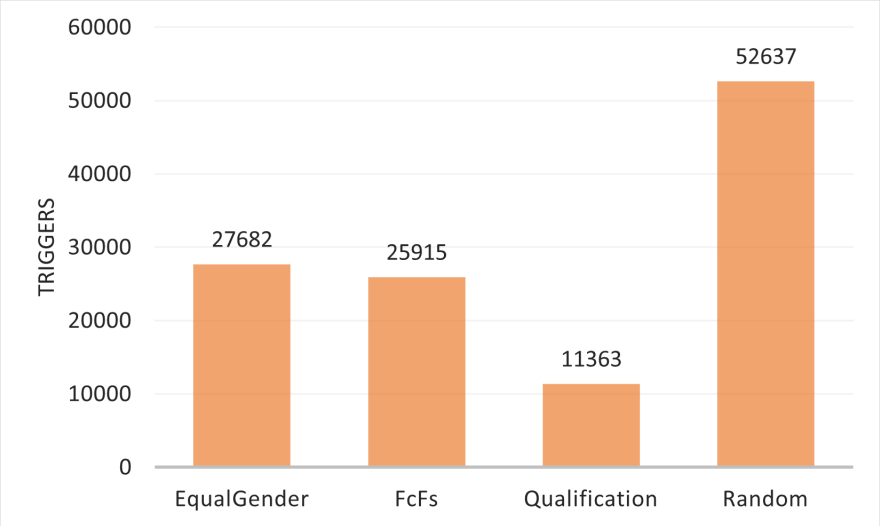


Figure 5.13: Number of Thrown Triggers with Counterfactual Fairness

same demographic group get different results, this does not count as unfair since these two are in the same demographic group. However, after that, every applicant with the same qualification and the other demographic group is counted as unfair because there is a person with the same qualification and a different gender that has a different outcome. The "Best Qualification" algorithm has by far the fewest triggers thrown. Since this algorithm tries to only accept applicants with the highest qualification, the condition for counterfactual fairness is fulfilled more often than with "Randomized". However, because this is supposed to be the reference algorithm, there are still raised too many triggers to label this algorithm fair according to counterfactual fairness.
Both other algorithms have between ~ 25900 and ~ 27700 triggers raised which can also be labeled unfair.

Concluding, in general, the relations are as expected. Relatively, "Best Qualification" is by far the fairest algorithm out of all according to counterfactual fairness, and "Randomized" is the unfairest. However, the absolute value of the number of triggers is too high to label any algorithm as fair. Since there is also no confidence or certain tolerance, it is not possible to adapt this fairness condition to get the proportions right.
With that, counterfactual fairness is not suitable for monitoring, since there was a modification required as stated in Section 3.4.3 to even be possible to express this fairness definition in RTLola. Further, even with this modification the result of the experiment shows that this specification of the fairness definition is unfitting and therefore not appropriate to monitor.

This section covered the analysis of the different decision-making algorithms in the environment of the specifications of the fairness conditions. For that, we analyzed the number of occurring triggers, investigated the timing of raised triggers, and examined sources of local unfairness. Further, we studied the different types of triggers, i.e., whether there are rising edges or the trigger is continuous leading to an unfair period.
Finally, we interpreted these results regarding the grade of unfairness within each algorithm but also the possibility of realizing the fairness definitions as RTLola specification.

Chapter 6
Related Work

6.1 Algorithmic Fairness

Further Fairness Definitions To analyze machine learning algorithms in more detail regarding fairness, other fairness conditions than the conditions used in this thesis were used . These conditions involve prediction attributes to achieve statements about the true positive rates (TPR) and the false positive rates (FPR). Moritz Hardt et al. [21] designed the fairness condition "Equalized Odds" to overcome the disadvantages of other fairness conditions. Equalized odds measure the difference between false positive rates and the difference between true positive rates of the different groups. A smaller difference between these rates indicates better fairness. The effectiveness of this condition is proven by the fact that with this condition, it was possible to detect bias in the COMPAS tool [24] that is mentioned in the introduction (Section 1).
Another fairness definition is "Equal opportunity", which requires true positive rates (TPR) to be similar across different groups [21]. The focus lying on only the true positive rates indicates one difference to equalized odds. The major difference, however, to equalized odds is the fact that equal opportunity indicates individual fairness. Other than demographic parity (Section 2.1.1), conditional statistical parity (Section 2.1.2), and equalized odds, equal opportunity focuses on fairness regarding individuals, as opposed to group fairness.
In conclusion, we saw that there exist more fairness conditions than the ones used in this thesis. However, we chose fairness definitions that do not require prediction tasks to gain binary results as output. Having binary results, it is possible to analyze the decision-making algorithm in more detail, since further computations can be acquired with the binary results.

Monitoring vs. Processing Mechanisms Over the past few years, the question of fairness in algorithms became more and more relevant. Especially the process of detecting unfairness and how to overcome the unfair part of these algorithms is still challenging according to Mehrabi [36]. In Machine learning, there were developed several methods

that impact the fairness of algorithms. in the following, we summarize the methods of achieving fair outputs concerning machine learning algorithms.
While monitoring can detect unfairness in algorithms based on the input and the output of an algorithm and a given specification, machine learning mechanisms however try to improve the result, meaning that the output will be fairer than it would be without the mechanism. Particularly three categories of mechanisms stand out, namely Pre-Processing, In-Processing, and Post-Processing. Pessach and Shmueli [40] explain Pre-Processing as changing the training data, before the algorithm gets the data as input, in the way that labels that could be treated wrong or in that sense unfair get changed so that the algorithm will treat them most likely fair [26]. Another way would be to modify the training data until the algorithm has problems differentiating between the two critical groups [19]. However, by modifying the input one cannot ensure that the output corresponds to the original training data i.e. the output lacks explainability. Another way of achieving fairer results is In-Processing, where the algorithm itself gets changed during its execution [1]. Additional penalties or regularization terms are added to e.g. satisfy the proxy for certain fairness conditions [27]. If, for some reason, it is not possible to perform Pre-Processing or In-Processing one can still get fairer results by applying Post-Processing mechanisms [12]. Here, Moritz Hardt et al. [22] suggest modifying the output by flipping some decisions to keep the fairness conditions alive. However one could argue that this technique is discriminatory for some individuals since their particular output could be flipped with the fairness conditions being the only reason.
Summarily monitoring observes unfairness in an "objective" way while the mechanisms described above try to improve the result regarding fairness by modifying specific parts of the whole process.

Formal Methods for Algorithmic Fairness A variety of different approaches for testing and verifying fairness have been invented. Udeshi et al. [45] analyzed the idea of designing an automated and directed testing technique to generate discriminatory inputs for models in the fields of machine learning. Further, Traimèr et al. [44] modeled the unwarranted-associations framework and instantiated it in FairTest. Another approach is given by Bastiani et al. [6] by using adaptive concentration inequalities to design a scalable sampling technique for providing fairness guarantees with the usage of probabilities for machine-learning models. Albarghouthi et al. [3] transformed fairness properties as probabilistic program properties and developed an SMT-based technique to verify fairness of decision-making programs. However, this technique only scales up to neural networks with at most 3 inputs and a single hidden layer with a maximum of 2 nodes. To overcome this issue a promising approach for analyzing fairness in neural networks is given in [46]. The approach combines forward and backward static analysis to certify fairness of feed-forward neural networks. With that, the forward pass is utilized to reduce the overall effort by dividing the input space of the neural network into independent partitions. The backward analysis then certifies fairness within each partition with respect to optional given sensitive attributes. This approach in the end yields which regions of the input space of the neural network are fair and which contain bias. With this approach, it is possible to analyze neural networks with significantly more hidden nodes.
To certify individual fairness like counterfactual fairness or equal opportunity mentioned in the further fairness definition paragraph, Rouss et al. [42] introduced a local property that coincides with robustness within a particular distance metric. Another approach is to repair biased decision-making programs with a program repair technique [2].

The approaches introduced above handle testing and verifying fairness in neural networks. However, by verifying networks the approaches are mostly unrealistic since they are based on a theoretical framework that is way too expensive regarding runtime. This thesis treats the algorithm (possibly a neural network) as a black box. With that, the size and runtime of the algorithm/neural network do not matter since with the monitoring approach the algorithm/neural network itself is not of interest. By only focussing on the input and output, we can monitor fairness conditions to (mostly) arbitrary algorithms/neural networks.

6.2 Monitoring

In the early stages of runtime monitoring, the system was mostly based on temporal logics [30, 33, 41]. With the rise of real-time temporal logics like MTL [29] and STL [35], a series of work introduced monitoring algorithms for real-time properties [5, 16].

Temporal Logics The tool named FoCs [13], developed by Haifa, is among others, a pioneer regarding translation from temporal logics to monitoring circuits. P2V [34] is a compiler that translates assertions written in sPSL [10] to Verilog code. BusMOP [39] synthesized monitor circuits from specifications written in past-time linear temporal logic for monitoring PCI bus traffic. Finally, MBAC [8] is an automata-based monitor synthesizer for PSL properties. Based on these constructions Jaksic et al. [23] introduced hardware runtime monitors for real-time properties, where monitors for STL specifications were implemented in an FPGA. This work expanded further with the tool R2U2 [37], an outline monitoring approach that allows specifications written in MTL, including future-time specifications, to be monitored.
Monitoring temporal logics brings the advantage of having formal guarantees on the space and time complexity of the synthesized monitor. Though, in cyber-physical systems, it is not enough to have properties that can only express absolute values like yes and no. For instance, monitoring LTL_3 the result is definitely either true, false, or unknown. The problem arising with this approach is having only absolute values, which lack in referring to more realistic scenarios. Using LTL_3 as logic in this thesis would state that an algorithm is either completely fair or unfair (or insufficient) with no respect so sensitive attributes. To monitor more realistic properties of certain systems, e.g. arithmetic operators are required.
With temporal logics it is possible to monitor the control flow of a program, however, it is not realistic to observe data in runtime. Since fairness is expressed in data, it is essential to observe the data. For that reason, the stream-based approach is used in this thesis rather than the temporal approach.

Stream-Based Language The specification-language RTLola [18] contains this expressiveness while still providing some formal guarantees. RTLola is a descriptive language, which subsumes temporal logics and can express both past and future properties. There are extensions to RTLola like TeSSLa and Striver. TeSSLa [11] introduces the possibility of monitoring piece-wise constant signals with streams emitting at different speeds with arbitrary latency. The major difference between RTLola and Striver [20] is that RTLola both offers variable-rate and fixed-rate streams and provides naive operators featuring sliding windows that translate between the two types of streams.

With this bundle of features and the combination of expressiveness and formal guarantees, RTLola is the specification language used in this thesis to investigate algorithmic fairness appropriately. RTLola is capable of observing and analyzing data, which is essential for investigating fairness conditions since they are based on data. Further, the stream-based approach brings the advantage of having fast outputs. At every time of the temporal input, it is possible to access the current state of fairness. In the runtime, there is no extra work to be done to extract intermediate results. With that, it is possible to investigate the behavior of decision-making algorithms in depth, since it provides insides into intermediate behavior.

Chapter 7
Conclusion

This thesis presented an RTLola-benchmark.This benchmark monitors decision-making algorithms regarding fairness. For that, we defined suitable fairness definitions regarding expressiveness and the feasibility of translating them into RTLola specifications. We then modified each fairness definition resulting in monitorable safety properties. These properties were then translated into RTLola specifications.
Additionally, we introduced different decision-making algorithms to compare them concerning fairness, and also to investigate the accuracy of the implemented fairness specifications. We defined an algorithm that ensures gender equality as a reference and also as a check of whether the fairness definitions are implemented correctly. Further, we introduced an algorithm that decides completely randomized and one that yields positive outcomes in its first decisions. These algorithms also serve as a check, because the expected result from them would be to decide unfairly. By monitoring them regarding fairness, we can see whether the fairness definitions can detect unfairness. Lastly, we introduced an algorithm that gives positive outcomes to the seemingly "best" applicants to investigate its behavior concerning fairness. To run a case study, we then generated artificial input scenarios which then are monitored regarding specific types of fairness.
Finally, we made a case study where we analyzed and interpreted the results of different experiments. We investigated the frequency of triggers, as well as the timing, i.e., the temporal occurrence of triggers. Furthermore, we compared the expected results with the accuracy of the results from the different fairness specifications. By that, the results for demographic parity (Definition 2.1.1), as well as for conditional statistical parity (Definition 2.1.2) were as expected. The algorithm that was expected to decide fair indeed yielded by far the best results compared to other algorithms. Further, we supported the statement that the fairness definitions are translated and implemented into RTLola specification by regarding the results of the experiments of an algorithm that decides completely randomized. With this algorithm, we wanted to achieve feedback about the accuracy of the specifications.
In our experiments, counterfactual fairness (Definition 2.1.3), however, is not suitable for monitoring with RTLola. The intended definition of this fairness definition is not realizable in RTLola and therefore needs a workaround. With that, the experiments showed unrealistically often unfairness, even with algorithms that seemingly ensure to fulfill the properties of counterfactual fairness.

7.1 Future Work

In future work, we aim to extend the specification language RTLola. In our computations for monitoring fairness, we require data from applicants. Because parts of the required data come earlier than others, these have to be stored until all computations with that user are executed. For that, we had to use a workaround where we use parameterized streams that store data from incoming applicants until they are not needed anymore. Instead, a database storing the data would be more practical. With that feature, a more adequate version of counterfactual fairness could be implemented, since it would be possible to compute the probability whether the same person belonging to a different demographic group would get a similar result. With these probabilities, it would be possible to argue whether individual decisions are made fairly according to counterfactual fairness.
Further, a request responding the number of parameterized versions of a stream would be beneficial. In this thesis, we needed to number of different seminars seen so far. To get this number, we had to parameterize over the seminar parameter and then always had to check if the seminar seen currently is new or already seen before. For a request that would yield a number of different versions of a parameterized stream, our workaround would be redundant.
We also want to compare our temporal approach with a database query. While our approach gives fast results, i.e. iterating through a file yields intermediate results, which has the advantage of revealing the behavior of a decision-making algorithm during processing. In contrast to that, with a database approach, one complete scenario could be analyzed faster, since this approach iterates over the complete input and then, in the end, checks for unfairness. However, analyzing intermediate results seems to be challenging with this approach.
Also, we want to compare our monitoring approach with other monitoring tools and evaluate the results.

Bibliography

[1] Alekh Agarwal, Alina Beygelzimer, Miroslav Dudík, John Langford, and Hanna M. Wallach. 2018. A Reductions Approach to Fair Classification. *CoRR* abs/1803.02453 (2018). http://arxiv.org/abs/1803.02453

[2] Aws Albarghouthi, Loris D'Antoni, and Samuel Drews. 2017a. Repairing Decision-Making Programs Under Uncertainty. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I (Lecture Notes in Computer Science)*, Rupak Majumdar and Viktor Kuncak (Eds.), Vol. 10426. Springer, 181–200. DOI:http://dx.doi.org/10.1007/978-3-319-63387-9_9

[3] Aws Albarghouthi, Loris D'Antoni, Samuel Drews, and Aditya V. Nori. 2017b. FairSquare: probabilistic verification of program fairness. *Proc. ACM Program. Lang.* 1, OOPSLA (2017), 80:1–80:30. DOI:http://dx.doi.org/10.1145/3133904

[4] Ricardo Baeza-Yates. 2018. Bias on the web. *Commun. ACM* 61, 6 (2018), 54–61. DOI:http://dx.doi.org/10.1145/3209581

[5] David A. Basin, Felix Klaedtke, Samuel Müller, and Eugen Zalinescu. 2015. Monitoring Metric First-Order Temporal Properties. *J. ACM* 62, 2 (2015), 15:1–15:45. DOI:http://dx.doi.org/10.1145/2699444

[6] Osbert Bastani, Xin Zhang, and Armando Solar-Lezama. 2019. Probabilistic verification of fairness properties via concentration. *Proc. ACM Program. Lang.* 3, OOPSLA (2019), 118:1–118:27. DOI:http://dx.doi.org/10.1145/3360544

[7] Rieke A Bogen, M. 2018. Help wanted: an examination of hiring algorithms, equity, and bias. *idk* (2018). https://www.upturn.org/work/help-wanted/

[8] Marc Boule and Zeljko Zilic. 2008. Automata-based assertion-checker synthesis of PSL properties. *ACM Trans. Design Autom. Electr. Syst.* 13, 1 (2008), 4:1–4:21. DOI:http://dx.doi.org/10.1145/1297666.1297670

[9] Joy Buolamwini and Timnit Gebru. 2018. Gender Shades: Intersectional Accuracy Disparities in Commercial Gender Classification. In *Conference on Fairness, Accountability and Transparency, FAT 2018, 23-24 February 2018, New York, NY, USA (Proceedings of Machine Learning Research)*, Sorelle A. Friedler and Christo Wilson (Eds.), Vol. 81. PMLR, 77–91. http://proceedings.mlr.press/v81/buolamwini18a.html

[10] Ping Hang Cheung and Alessandro Forin. 2007. A C-Language Binding for PSL. In *Embedded Software and Systems, [Third] International Conference, ICESS 2007, Daegu, Korea, May 14-16, 2007, Proceedings (Lecture Notes in Computer Science)*, Yann-Hang Lee, Heung-Nam Kim, Jong Kim, Yongwan Park, Laurence Tianruo Yang, and Sung Won Kim (Eds.), Vol. 4523. Springer, 584–591. DOI:http://dx.doi.org/10.1007/978-3-540-72685-2_54

[11] Lukas Convent, Sebastian Hungerecker, Torben Scheffel, Malte Schmitz, Daniel Thoma, and Alexander Weiss. 2018. Hardware-Based Runtime Verification with Embedded Tracing Units and Stream Processing. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 43–63. DOI:http://dx.doi.org/10.1007/978-3-030-03769-7_5

[12] Sam Corbett-Davies, Emma Pierson, Avi Feller, Sharad Goel, and Aziz Huq. 2017. Algorithmic decision making and the cost of fairness. *CoRR* abs/1701.08230 (2017). http://arxiv.org/abs/1701.08230

[13] Anat Dahan, Daniel Geist, Leonid Gluhovsky, Dmitry Pidan, Gil Shapir, Yaron Wolfsthal, Lyes Benalycherif, Romain Kamdem, and Younes Lahbib. 2005. Combining System Level Modeling with Assertion Based Verification. In *6th International Symposium on Quality of Electronic Design (ISQED 2005), 21-23 March 2005, San Jose, CA, USA*. IEEE Computer Society, 310–315. DOI:http://dx.doi.org/10.1109/ISQED.2005.32

[14] Ben D'Angelo, Sriram Sankaranarayanan, César Sánchez, Will Robinson, Bernd Finkbeiner, Henny B. Sipma, Sandeep Mehrotra, and Zohar Manna. 2005. LOLA: Runtime Monitoring of Synchronous Systems. In *12th International Symposium on Temporal Representation and Reasoning (TIME 2005), 23-25 June 2005, Burlington, Vermont, USA*. IEEE Computer Society, 166–174. DOI:http://dx.doi.org/10.1109/TIME.2005.26

[15] David Danks and Alex John London. 2017. Algorithmic Bias in Autonomous Systems. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017, Melbourne, Australia, August 19-25, 2017*, Carles Sierra (Ed.). ijcai.org, 4691–4697. DOI:http://dx.doi.org/10.24963/ijcai.2017/654

[16] Jyotirmoy V. Deshmukh, Alexandre Donzé, Shromona Ghosh, Xiaoqing Jin, Garvit Juniwal, and Sanjit A. Seshia. 2017. Robust online monitoring of signal temporal logic. *Formal Methods Syst. Des.* 51, 1 (2017), 5–30. DOI:http://dx.doi.org/10.1007/s10703-017-0286-7

[17] Cynthia Dwork, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Richard S. Zemel. 2011. Fairness Through Awareness. *CoRR* abs/1104.3913 (2011). http://arxiv.org/abs/1104.3913

[18] Peter Faymonville, Bernd Finkbeiner, Maximilian Schwenger, and Hazem Torfah. 2017. Real-time Stream-based Monitoring. *CoRR* abs/1711.03829 (2017). http://arxiv.org/abs/1711.03829

[19] Sorelle A. Friedler, Carlos Scheidegger, and Suresh Venkatasubramanian. 2014. Certifying and removing disparate impact. *CoRR* abs/1412.3756 (2014). http://arxiv.org/abs/1412.3756

[20] Felipe Gorostiaga and César Sánchez. 2018. Striver: Stream Runtime Verification for Real-Time Event-Streams. In *Runtime Verification - 18th International Conference, RV 2018, Limassol, Cyprus, November 10-13, 2018, Proceedings (Lecture Notes in Computer Science)*, Christian Colombo and Martin Leucker (Eds.), Vol. 11237. Springer, 282–298. DOI:http://dx.doi.org/10.1007/978-3-030-03769-7_16

[21] Moritz Hardt, Eric Price, and Nati Srebro. 2016a. Equality of Opportunity in Supervised Learning. In *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett (Eds.). 3315–3323. https://proceedings.neurips.cc/paper/2016/hash/9d2682367c3935defcb1f9e247a97c0d-Abstract.html

[22] Moritz Hardt, Eric Price, and Nathan Srebro. 2016b. Equality of Opportunity in Supervised Learning. *CoRR* abs/1610.02413 (2016). http://arxiv.org/abs/1610.02413

[23] Stefan Jaksic, Ezio Bartocci, Radu Grosu, Reinhard Kloibhofer, Thang Nguyen, and Dejan Nickovic. 2015. From signal temporal logic to FPGA monitors. In *13. ACM/IEEE International Conference on Formal Methods and Models for Codesign, MEMOCODE 2015, Austin, TX, USA, September 21-23, 2015*. IEEE, 218–227. DOI: http://dx.doi.org/10.1109/MEMCOD.2015.7340489

[24] Lauren Kirchner Julia Angwin Jeff Larson, Surya Mattu. 2016. How We Analyzed the COMPAS Recidivism Algorithm. (2016). https://www.propublica.org/article/how-we-analyzed-the-compas-recidivism-algorithm

[25] Matthew Joseph, Michael J. Kearns, Jamie Morgenstern, and Aaron Roth. 2016. Fairness in Learning: Classic and Contextual Bandits. *CoRR* abs/1605.07139 (2016). http://arxiv.org/abs/1605.07139

[26] Calders T. Kamiran, F. 2012. Data preprocessing techniques for classification without discrimination. (2012). 0.1007/s10115-011-0463-8

[27] Toshihiro Kamishima, Shotaro Akaho, Hideki Asoh, and Jun Sakuma. 2012. Fairness-Aware Classifier with Prejudice Remover Regularizer. In *Machine Learning and Knowledge Discovery in Databases - European Conference, ECML PKDD 2012, Bristol, UK, September 24-28, 2012. Proceedings, Part II (Lecture Notes in Computer Science)*, Peter A. Flach, Tijl De Bie, and Nello Cristianini (Eds.), Vol. 7524. Springer, 35–50. DOI:http://dx.doi.org/10.1007/978-3-642-33486-3_3

[28] Jon M. Kleinberg, Sendhil Mullainathan, and Manish Raghavan. 2016. Inherent Trade-Offs in the Fair Determination of Risk Scores. *CoRR* abs/1609.05807 (2016). http://arxiv.org/abs/1609.05807

[29] Ron Koymans. 1990. Specifying Real-Time Properties with Metric Temporal Logic. *Real Time Syst.* 2, 4 (1990), 255–299. DOI:http://dx.doi.org/10.1007/BF01995674

[30] Orna Kupferman and Moshe Y. Vardi. 2001. Model Checking of Safety Properties. *Formal Methods Syst. Des.* 19, 3 (2001), 291–314. DOI:http://dx.doi.org/10.1023/A:1011254632723

[31] Matt J. Kusner, Joshua R. Loftus, Chris Russell, and Ricardo Silva. 2017. Counterfactual Fairness. *CoRR* abs/1703.06856 (2017). http://arxiv.org/abs/1703.06856

[32] Leslie Lamport. 1979. A New Approach to Proving the Correctness of Multiprocess Programs. *ACM Trans. Program. Lang. Syst.* 1, 1 (1979), 84–97. DOI:http://dx.doi.org/10.1145/357062.357068

[33] Insup Lee, Sampath Kannan, Moonjoo Kim, Oleg Sokolsky, and Mahesh Viswanathan. 1999. Runtime Assurance Based On Formal Specifications. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA 1999, June 28 - Junlly 1, 1999, Las Vegas, Nevada, USA*, Hamid R. Arabnia (Ed.). CSREA Press, 279–287.

[34] Hong Lu and Alessandro Forin. 2007. *The Design and Implementation of P2V, An Architecture for Zero-Overhead Online Verification of Software Programs*. Technical Report MSR-TR-2007-99. 12 pages.

[35] Oded Maler and Dejan Nickovic. 2004. Monitoring Temporal Properties of Continuous Signals. In *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings (Lecture Notes in Computer Science)*, Yassine Lakhnech and Sergio Yovine (Eds.), Vol. 3253. Springer, 152–166. DOI:http://dx.doi.org/10.1007/978-3-540-30206-3_12

[36] Ninareh Mehrabi, Fred Morstatter, Nripsuta Saxena, Kristina Lerman, and Aram Galstyan. 2021. A Survey on Bias and Fairness in Machine Learning. *ACM Comput. Surv.* 54, 6 (2021), 115:1–115:35. DOI:http://dx.doi.org/10.1145/3457607

[37] Patrick Moosbrugger, Kristin Y. Rozier, and Johann Schumann. 2017. R2U2: monitoring and diagnosis of security threats for unmanned aerial systems. *Formal Methods Syst. Des.* 51, 1 (2017), 31–61. DOI:http://dx.doi.org/10.1007/s10703-017-0275-x

[38] Amitabha Mukerjee, Rita Biswas, Kalyanmoy Deb, and Amrit P. Mathur. 2002. Multi–objective Evolutionary Algorithms for the Risk–return Trade–off in Bank Loan Management. *International Transactions in Operational Research* 9 (2002), 583–597.

[39] Rodolfo Pellizzoni, Patrick O'Neil Meredith, Marco Caccamo, and Grigore Rosu. 2008. Hardware Runtime Monitoring for Dependable COTS-Based Real-Time Embedded Systems. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*. IEEE Computer Society, 481–491. DOI:http://dx.doi.org/10.1109/RTSS.2008.43

[40] Dana Pessach and Erez Shmueli. 2020. Algorithmic Fairness. *CoRR* abs/2001.09784 (2020). https://arxiv.org/abs/2001.09784

[41] Amir Pnueli. 1977. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*. IEEE Computer Society, 46–57. DOI:http://dx.doi.org/10.1109/SFCS.1977.32

[42] Anian Ruoss, Mislav Balunovic, Marc Fischer, and Martin T. Vechev. 2020. Learning Certified Individually Fair Representations. *CoRR* abs/2002.10312 (2020). https://arxiv.org/abs/2002.10312

[43] Harini Suresh and John V. Guttag. 2019. A Framework for Understanding Unintended Consequences of Machine Learning. *CoRR* abs/1901.10002 (2019). http://arxiv.org/abs/1901.10002

[44] Florian Tramèr, Vaggelis Atlidakis, Roxana Geambasu, Daniel J. Hsu, Jean-Pierre Hubaux, Mathias Humbert, Ari Juels, and Huang Lin. 2017. FairTest: Discovering Unwarranted Associations in Data-Driven Applications. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*. IEEE, 401–416. DOI:http://dx.doi.org/10.1109/EuroSP.2017.29

[45] Sakshi Udeshi, Pryanshu Arora, and Sudipta Chattopadhyay. 2018. Automated directed fairness testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 98–108. DOI:http://dx.doi.org/10.1145/3238147.3238165

[46] Caterina Urban, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. 2020. Perfectly parallel fairness certification of neural networks. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 185:1–185:30. DOI:http://dx.doi.org/10.1145/3428253

[47] Sahil Verma and Julia Rubin. 2018. Fairness definitions explained. In *Proceedings of the International Workshop on Software Fairness, FairWare@ICSE 2018, Gothenburg, Sweden, May 29, 2018*, Yuriy Brun, Brittany Johnson, and Alexandra Meliou (Eds.). ACM, 1–7. DOI:http://dx.doi.org/10.1145/3194770.3194776

[48] James Zou and Londa Schiebinger. 2018. AI can be sexist and racist it's time to make it fair. *Nature Publishing Group* (2018).