What Leads to a Confirmatory or Disconfirmatory Behaviour of Software Testers?

Iflaah Salman, Pilar Rodríguez, Burak Turhan, *Member, IEEE,* Ayşe Tosun, *Member, IEEE,* and Arda Güreller

Abstract—**Background:** The existing literature in software engineering reports adverse effects of confirmation bias on software testing. Confirmation bias among software testers leads to confirmatory behaviour, which is designing or executing relatively more specification consistent test cases (confirmatory behaviour) than specification inconsistent test cases (disconfirmatory behaviour). **Objective:** We aim to explore the antecedents to confirmatory and disconfirmatory behaviour of software testers. Furthermore, we aim to understand why and how those antecedents lead to (dis)confirmatory behaviour. **Method:** We follow grounded theory method for the analyses of the data collected through semi-structured interviews with twelve software testers. **Results:** We identified twenty antecedents to (dis)confirmatory behaviour, and classified them in nine categories. Experience and Time are the two major categories. Experience is a disconfirmatory category, which also determines which behaviour (confirmatory or disconfirmatory) occurs first among software testers, as an effect of other antecedents. Time Pressure is a confirmatory antecedent of the Time category. It also contributes to the confirmatory effects of antecedents of other categories. **Conclusion:** The disconfirmatory antecedents, especially that belong to the testing process, e.g., test suite reviews by project team members, may help circumvent the deleterious effects of confirmation bias in software testing. If a team's resources permit, the designing and execution of a test suite could be divided among the test team members, as different perspectives of testers may help to detect more errors. The results of our study are based on a single context where dedicated testing teams focus on higher levels of testing. The study's scope does not account for the testing performed by developers. Future work includes exploring other contexts to extend our results.

Index Terms—Software Testing, Cognitive Biases, Confirmation Bias, Grounded Theory, Interviews.

1 Introduction

Confirmation bias is the cognitive tendency to look for evidence that confirms, rather than refutes, one's prior beliefs [1]. In software testing, confirmation bias occurs when developers or testers exercise a program with the data that is consistent with its specified behaviour instead of inconsistent data [2]. Confirmation bias leads to confirmatory behaviour by software testers during testing [3]. For example, if requirements specification state that *...the phone number field accepts seven digits from* 0 *to* 9; a consistent test case would validate the behaviour of the field by providing in, e.g., 0123456 as an input test data. An inconsistent test case would validate the field's behaviour with inconsistent data, e.g., a - a letter instead of a digit.

The higher the level of confirmation bias, the more adverse effects it has on software testing [4], [5], [6], [7]. For example, Çalikli and Bener observed a positive correlation between software defect density and confirmation bias levels of software developers [4], [8]. In their experiments, Teasley et al. observed that participants designed two to four times more positive test cases compared to negative test cases[1] (i.e., confirmation bias) [7]. Similarly, Causevic et al. also found a significant difference between the number of positive and negative test cases designed by the participants in an experimental study on test-driven development [9]. Salman et al.'s work also supports these findings, in their experiment, participants designed significantly more consistent test cases, with respect to provided specifications, compared to inconsistent test cases in performing functional testing [3].

Mohanani et al. found the primary studies that investigated the effects and antecedents to confirmation bias in software testing were all experiments [10]. For example, an experimental study observed that a lack of logical reasoning skills is an antecedent to confirmation bias [10]. The authors identified a need to conduct more qualitative research that explores how cognitive biases are manifested in the software engineering (SE) industry rather than only focusing on causal relationships. The primary studies on antecedents to confirmation bias in software testing are limited in providing an insight into the phenomenon [10]. These findings, therefore, establish a need of not only to explore more antecedents to confirmation bias but also to understand why and how it occurs among software testers.

The goal of this paper is to explore antecedents that

- *I. Salman is with M3S Group, University of Oulu, Oulu, Finland. E-mail: iflaah.salman@oulu.fi*
- *P. Rodríguez is with M3S Group, University of Oulu, Oulu, Finland, and Escuela Tecnica Superior de Ingenieros Informaticos, Universidad Politecnica de Madrid, Madrid, Spain. E-mail: pilar.rodriguez@upm.es*
- *B. Turhan is with Faculty of Information Technology, Monash University, VIC, Australia, and M3S Group, University of Oulu, Oulu, Finland. E-mail: burak.turhan@monash.edu*
- *A. Tosun is with Faculty of Computer and Informatics Engineering, Istanbul Technical University, Istanbul, Turkey. E-mail: tosunay@itu.edu.tr*
- *A. Güreller is with Ericsson, Istanbul, Turkey. Email: arda.gureller@ericsson.com*

1. Positive/negative test case is another terminology for what we call consistent/inconsistent test case. We use the latter one for the remainder of this paper.

may lead to confirmation bias in software testing. The antecedents may belong to the working environment, could be part of the testing process or are personal attributes of software testers. We additionally aim to understand why and how these antecedents lead software testers to confirmatory behaviour. In order to address our objectives, we apply the Glaserian grounded theory to explore the phenomenon of confirmation bias among software testers. We conducted twelve semi-structured interviews with software testers to collect our data. They were all employees of the same company but worked in different projects.

We identified nine antecedents to confirmatory behaviour and eight to disconfirmatory behaviour. A disconfirmatory behaviour is contrasting to a confirmatory behaviour, i.e., it may mitigate confirmation bias. Additionally, three more antecedents were found that lead to both (confirmatory and disconfirmatory) behaviours by software testers. *Both* refers to the completeness of a test suite[2] that may also mitigate confirmation bias. Experience of testing in general and particular to the project are two major disconfirmatory antecedents. They determine the confirmatory or disconfirmatory behaviour of testers due to other antecedents. Project's testing experience also improves the completeness of a test suite. Time pressure is a major confirmatory antecedent for software testers. It contributes to promoting the confirmatory influence of other confirmatory antecedents. For example, time pressure promotes the confirmatory behaviour of a tester in case of a minor functional change (a confirmatory antecedent).

Our study contributes by generating a grounded theory that explains the phenomenon of confirmation bias among software testers. We also contribute with the identification of thirteen new antecedents, relative to the existing ones in the SE literature, that may lead to confirmation bias. We also provide a list of disconfirmatory antecedents that can be used by practitioners to alleviate confirmation bias.

Section 2 presents the related work and the conceptual background. Research Method is detailed in Section 3. The results are presented in Section 4, and discussed along with the validity threats in Section 5. Section 6 concludes the paper.

2 Related Work and Background

Humans rely on simplifying heuristics for judgement of uncertain events instead of relying on formal logic [11]. These heuristics usually offer a workable solution, but may also lead to systematic errors in decision making, known as cognitive biases [11], [12]. The concept of cognitive biases was first introduced by Tversky and Kahneman in the early 1970s, and is defined as, *"cognitive biases are cognitions or mental behaviours that prejudice decision quality in a significant number of decisions for a significant number of people"* [1, p. 59], [10], [12]. Cognitive biases are also referred to as judgement biases or decision biases [13]. The human mind is inherent to cognitive biases [1], [13], [14]. Kahneman et al. elaborate on why humans are incapable to recognise their own cognitive biases by referring to two modes of thinking; system-one and system-two, also referred to as the dual-process theory [14], [15]. System-one intuitive, thinking is fast and effortless, which makes it more prone to biases [10], [16]. System-two reflective, thinking is effortful, intentional and slow, therefore, less prone to biases [10], [16]. Thoughts are usually determined by system-one, humans are unaware of it because it is proficient in its operation [15], [16]. In addition to system-one, noisy information processing, emotion and social influences can also generate cognitive biases [10]. Cognitive biases also form *biasplexes* because some biases may overlap, interact and reinforce each other [17]. Therefore, when biases occur it is uncertain which one is the cause and which one is the effect [10].

Confirmation bias is a cognitive bias [1], [11]. Mohanani et al. categorise confirmation bias to the category of interest biases among the categories defined for cognitive biases in the SE discipline [10]. Confirmation bias negatively affects multiple areas of SE, e.g., maintenance [18], design [19] and testing [2], [20]. The software testing studies that investigated confirmation bias use different terminologies, e.g., positive test bias, but essentially refer to the same phenomenon of confirmation bias [3], [21].

The earliest (1993 1994) work exploring the impact of confirmation bias in software testing was conducted by Leventhal et al. and Teasley et al. [2], [6], [7]. These authors, in their family of experiments conducted with advanced testers (senior-level and graduate students in computer science), observed multiple factors that may cause the manifestation of positive test bias in functional software testing. The results showed that a higher level of expertise and completeness of specifications may cause less positive test bias [6], [7]. Another studied factor, error feedback (the effect of presence or absence errors), was not confirmed to cause the effect possibly due to the types of software used in the experiments [6].

The second era of focus begins from 2010 when multiple studies examined the effects of confirmation bias in software testing. Çalikli and Bener [5] and Çalikli et al. [22], in their series of experiments, assessed confirmation bias levels of the participants by deriving measures from psychological instruments, Wason's Rule Discovery and Selection Task. In an experiment with software engineers and graduate students, Çalikli et al. found that company culture affected the confirmation bias levels [23]. The authors also investigated the effects of logical reasoning skills acquired through education, experience and activeness in testing and development, job titles (tester, developer, analyst, researcher), development methods, company size (large, small and medium enterprises), educational background (undergraduate) and educational level (bachelor's, master's) [5], [8]. They found that confirmation bias levels were low due to logical reasoning skills and for those participants who were experienced but inactive in testing or development [5], [8]. The rest of the factors were not observed to affect confirmation bias levels except the job title - researcher [5], [8]. Çalikli and Bener related the lower levels of confirmation bias of researchers to their critical and analytical skills [5].

In a test-driven development (TDD) experiment, carried out in the industry, Causevic et al. observed that participants created more positive test cases compared to negative test cases [9]. The experimenters also observed that negative test

2. It refers to the completeness of design/execution in terms of consistent and inconsistent test cases.

cases have a higher tendency of finding defects compared to positive test cases [9]. Eldh investigated whether negative testing reveals 'real important faults' of the system under test (SUT) by applying negative testing techniques referred to as 'attacks' by Whittaker et al. [24], [25]. The author found that negative testing could not find any major faults for the SUT, albeit notable ones [24]. Eldh attributed the findings to the high quality of the SUT and the types of the executed negative test cases [24].

According to Salman et al., confirmation bias occurs when a software tester designs relatively more *consistent* test cases (consistent with the requirements specification) in comparison to *inconsistent* test cases [3]. An *inconsistent* test case validates a behaviour of the software application that is not explicit in the requirements specification or is an outside-of-the-box test case, within the context of the SUT [3]. In functional test case design, a consistent test case is an indication of a confirmatory behaviour; similarly, an inconsistent test case indicates a disconfirmatory behaviour on a tester's end [3].

Multiple qualitative studies on cognitive biases have used interview data collection method for grounded theory and case studies. The objectives were to identify the occurrence of cognitive biases in the studied context, antecedents to cognitive biases and mitigation techniques for cognitive biases [26], [27], [28], [29], [30]. For example, Cunha et al. conducted semi-structured interviews for a cross-case analysis of decision-making in project management [26]. The authors identified antecedents to multiple cognitive biases, e.g., the absence of records of the learned lessons from previous projects, can lead to availability bias[3] during decision making by a project manager [26]. These studies support the use of interview data collection method and grounded theory as an appropriate approach for exploring and understanding the phenomenon of confirmation bias.

This section shows that the existing literature on confirmation bias is limited to controlled experiments only. These studies tested hypotheses about isolated factors as potential antecedents to confirmation bias. A qualitative study is, therefore, required to explore what other antecedents lead to confirmation bias and how? Our study, by applying grounded theory, explores other antecedents to confirmation bias in software testing. We also aim to understand how these antecedents lead to confirmation bias. The postulates generated by our theory can be verified by further empirical studies.

3 Research Method

We apply grounded theory (GT) as our research method. The objective of GT is to generate a theory that is grounded in data [31]. According to Urquhart, *"Theory asserts a plausible relationship between concepts and sets of concepts, and the resulting theory can be reported in a narrative framework or a set of propositions"* [31, p. 5]. GT is suitable to address our study's objectives because of a lack of empirical evidence on the antecedents to confirmation bias, and why testers manifest confirmation bias is yet unknown; to the best of our knowledge [10]. Therefore, we try to understand, "What's going on here?" [32, p.120]. "Here", refers to our context of understanding, why and how confirmation bias occurs. We apply the Glaserian version of GT because we wanted the specific research questions to emerge during the data analysis [32]. Our ontological position is positivism. By using the GT's inductive theory-building process, we first present the theory of the phenomenon under study in narrative form in Section 4. An integrative diagram of the theory is then presented in Section 5 that explains the inter-relationship of the derived concepts and categories. We follow the guidelines by Stol et al. for reporting this study [32].

3. "Availability bias refers to a tendency of being influenced by the information that is easy to recall and by the information that is recent or widely publicised" [10, p.21].

3.1 Goal

In the context of this study, confirmatory behaviour occurs when a tester designs or executes consistent test case(s), and a disconfirmatory behaviour otherwise. In order to have complete coverage for the SUT, a tester should manifest both confirmatory and disconfirmatory behaviours. In other words, a test suite should be complete in terms of consistent and inconsistent test cases. We refer to it as the completeness of a test suite in this study.

Certain antecedents may lead to a compromise of one behaviour over the other, e.g., disconfirmatory over confirmatory. Thus, possibly not only promoting confirmation bias but also limiting the completeness of a test suite. The objective of this study is to explore the antecedents to the confirmatory and disconfirmatory behaviour of software testers while performing testing. It is worth to find out antecedents also for disconfirmatory behaviour because the absence of them may imply the promotion of confirmatory behaviour, which may lead to confirmation bias by software testers. We, therefore, answer the following research questions:

RQ1: What are the antecedents to confirmatory and disconfirmatory behaviour among software testers?

RQ2: Why or how do the antecedents influence the behaviour of software testers as confirmatory or disconfirmatory?

The objective of the study initially helped define RQ1. RQ2 emerged during the data analysis, which also, in turn, refined RQ1. The application of the Glaserian coding techniques enabled us to break down the research question into specific research questions [31].

3.2 Context, Participants and Data Collection

We used interviews as a primary data collection method for our study. Interviews are a qualitative source of data that perfectly aligns with the inductive process of GT [31].

We interviewed twelve professionals working in a world leading company in Information and Communication Technology domain. To maintain anonymity, we refer to this company as Company-ICT in our study. The Company-ICT is offering services in networks, digitisation of solutions, managing IT services and providing solutions in IoT areas. The Company-ICT develops internal software projects using Agile software development. The projects have dedicated testing teams who perform higher levels of testing, e.g., integration testing, while developers are responsible for

performing unit testing. In case of a small project team, one person may perform multiple roles, despite their job title, e.g., software architect also performs testing when required. The company also has dedicated test automation teams that are not part of any particular project; they automate manual test suites of the projects. We chose this company for two main reasons: 1) it acts as a vendor to conduct system tests on behalf of its business contractors, thus, 2) it has been collaborating with academia, international partners in EU and nationally funded projects for improving its testing process, test effectiveness and measurement.

The interviewed professionals participated in this study voluntarily. We specified the recruitment of test engineers or engineers with testing experience to our contact person at the company because we wanted a sample well aligned with the goal of our study. The champion approached the pool of more than fifty software testing engineers through their respective managers. Twelve engineers positively responded to the call of the champion for participation. The sample consisted of 10 test engineers. The additional two were: a solution architect and a software engineer. The solution architect was also partly performing activities as a test engineer, and the software engineer was involved in both development and testing (as a test engineer). The participants belong to different projects or domains at the company's two sites. We refer to all these participants as testers from now onward in this study. Based on the characteristics of our participants and set-up of the company, testing performed as a developer is not accounted for in this study's scope. We focus on the higher levels of testing performed by testers. It is important to note that our study does not aim to achieve statistical generalisation with this sample because, in qualitative research, researchers generalise to theory instead of a population [31], [33]. We are exploring the phenomenon in the defined context rather than achieving representativeness [33]. However, the issue of achieving generalisability with a positivist GT approach is discussed later in Section 5.5.

The format of the interviews was semi-structured. Before conducting the actual interviews, we piloted the script with a software engineer from a different company. The objective of the pilot interview was to improve the wording of the questions and timing of the session. The interviews were conducted in October 2017 via Skype through video-calls, and voice-calls when the video was not viable. On average, it took 65 min per person to interview. We collected approximately 13 hr of audio (with informed consent) and 127 pages of verbatim transcribed data. One of the authors went through the transcriptions and audio files again to tally the content and to ensure that technical terms were correctly transcribed. The interview script is available as an online appendix[4].

The characterisation of participants is given in Table 1. The participants have at least two years of working experience at the Company-ICT, except for two of them. Only one participant has only 6 months of testing experience otherwise the average testing experience is approx. 6 years. Two of the testers are automation test engineers, one is performing testing both manually and in an automated way, the rest are all manual testers.

3.3 Data Analysis

The data analysis procedures in GT are systematic [31]. The applied coding techniques are open coding, selective coding and theoretical coding. The application of the constant comparison method (CCM) to the coding techniques made coding an iterative process [31], [32], [34]. We followed the guidelines by Urquhart and Boeije for applying the mentioned techniques [31], [34]. An example of deriving two of the antecedents (past experience, project experience) belonging to a single category, *experience*, through the applied coding techniques is illustrated in Figure 1. The sample raw data from three interviews, P2, P3 and P10, is shown separately in the figure. We first applied open coding to the individual interviews and then filtered it to the relevant concepts, i.e., antecedents, per the objective of our study. The open coding is shown as **bold** texts in the excerpts. After the application of CCM within the interviews and among the interviews, and the application of selective coding, concepts emerged. The emerged concepts were then grouped under a category, which is a higher level of abstraction. In the illustration - Figure 1, the category is *experience*, which is one of the identified antecedents to disconfirmatory behaviour. We also applied memoing and memo sorting along the process of selective coding and CCM. It enabled us to classify the antecedents as (dis)confirmatory and to capture the relationships between the emerging concepts. This led us to the integrative diagram as a result of theoretical coding, which is the third stage of coding in the Glaserian GT [31], [32]. We used NVivo[5] data analysis tool for coding.

We implement coding validity steps because of our positivist ontological position, as recommended by Urquhart [31]. In our context, *intercoder reliability* refers to, two or more coders identify the same code (antecedent) and use same classification (e.g., confirmatory, disconfirmatory) for the code, when coding independently [35]. *Intercoder agreement* assurance requires that the coders discuss and reconcile their coding discrepancies [35]. We performed multiple steps to ensure intercoder reliability and agreement for the identification of antecedents to confirmatory or disconfirmatory behaviour.

One of the authors (the interviewer) initially formed a list of the terms that interviewees used to indicate their confirmatory or disconfirmatory behaviour during testing. Afterwards, we developed a coding protocol that comprised coding guidelines and the previously formed list of terms. One of the authors is experienced in applying grounded theory coding techniques, two of the authors contributed with knowledge on software testing and cognitive biases. One of the authors brought in expertise in software testing. Therefore, we also developed a unanimous understanding of confirmation bias and its manifestation in the studied context. The four authors then performed a pilot coding of a randomly chosen interview (*P2*) from the set of twelve interviews. The objective was to identify antecedents and classify them as confirmatory or disconfirmatory, and validate the coding process. The objective was also to ensure that codes

4. http://doi.org/10.5281/zenodo.3376920

5. https://www.qsrinternational.com/nvivo

TABLE 1
Participants (**Exp. in Testing** does not account the duration of testing as a software developer)

P#	Job Title	Exp. in Company-ICT	Exp. in Testing	Tester Type	Interview Length
P1	Software Engineer	2 yr 11 mos	6 mos	Manual	71 min
P2	Test Engineer	6 yr	7 yr	Manual	79 min
P3	Senior IT Test Engineer	5 yr	9 yr 6 mos	Manual, Automation	57 min
P4	Senior Software Test Engineer	2 yr	2 yr	Manual	80 min
P5	Test Engineer	5 mos	9 yr	Manual	58 min
P6	Solution Architect	5 yr 6 mos	12 yr	Automation	76 min
P7	Experienced Integration Engineer	4 yr	4 yr	Manual	81 min
P8	Integration Engineer	2 yr	2 yr	Automation	58 min
P9	Configuration Manager & Test Engineer	4 yr	6 yr	Manual	68 min
P10	IT System Expert (Software Test Expert)	3 yr	6 yr	Manual	51 min
P11	Test Team Manager	5 yr 6 mos	5 yr 6 mos	Manual	53 min
P12	Software Test Engineer	1 yr	4 yr 6 mos	Manual	39 min

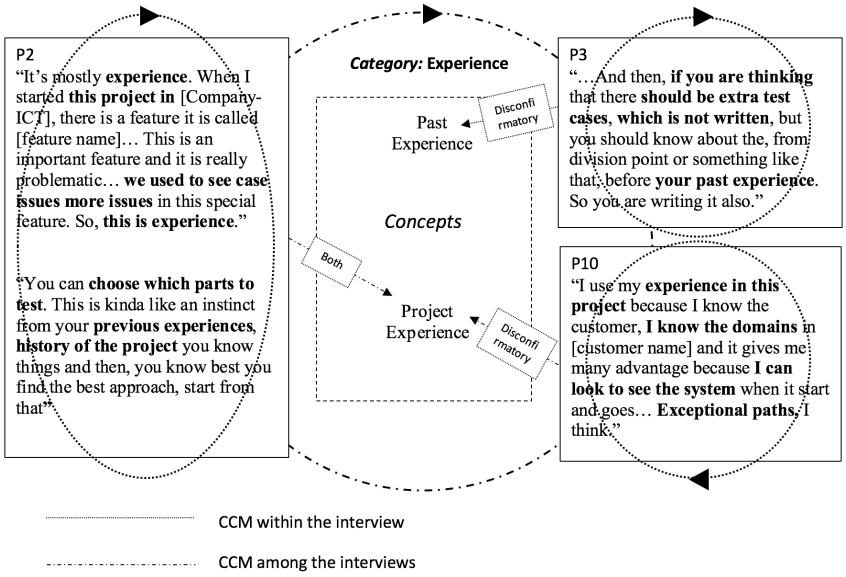

Fig. 1. GT Coding Mechanisms

produced by any single knowledgeable coder would be reproducible by other equally knowledgeable coders, as all the authors may not be available to code the data [35].

The joint discussion session, after the pilot coding, revealed a need for a refined understanding of an antecedent and introduction of more categories for classifying antecedents. As, we noticed that some of the antecedents could neither be classified as confirmatory nor disconfirmatory, rather both or unknown - see Table 2. Additionally, we decided to disregard the data where the interviewee's understanding of the terms differed from the theoretical definitions, e.g., referring functional testing to as non-functional testing. This decision was taken to prevent personal interpretations on the coder's behalf because further communication with the participant also couldn't clarify those misunderstandings. The percentage of such data accumulated to 0.92% by the end of the data analysis of all interviews.

Due to the recognition of the issue with antecedents' classification, we assessed intercoder reliability[6] of our pilot coding only for the identified antecedents. The agreement between (coder1, coder2) was 54%, (coder1, coder3) was 39% and (coder1, coder4) was 46%. We assessed agreements in pairs with coder1 because it was decided that coder1 would code the rest of the interviews. In order to improve the intercoder reliability measure, we performed the following steps.

The other three coders then revised their identified antecedents and reclassified them according to the classification presented in Table 2. It was followed by one-on-one discussions with coder1 on the identified antecedents and their classification. For example, between coder1 and coder2; a code was defined for higher authority involvement in deciding the density of issues to deliver a patch with. After analysing the excerpts and context, it was decided that this does not qualify as an antecedent because it is not influencing a tester's behaviour during testing activities. Later, coder3 and coder4 also agreed to that decision. These discussions provided input to coder1 to revise the coding of the *P2* interview. Coder1 then coded the rest of the interviews in three iterations. After each iteration, we held a joint discussion session in which confusions regarding the coding were resolved. For example, a confusing excerpt that

6. Percentage of the total number of common antecedents divided by the total number of identified antecedents.

was coded for the antecedent *change request* was resolved to illegible evidence towards any of the classification: *"Actually, we make a task separation between us [with his colleague] before making test cases, and we start to create test cases of our products. And, then, we come together and we try to question our test cases, we try to reveal the solution documents together and we try to understand, actually, we have a short [technique name]. We are trying to compare our test cases with these objectives, smart. And, we are trying to meet the requirements which we gathered. There is no specific thing like we do. We're just discussing." - P7*. This was an answer to a question about the participant's discussion with their colleague, which did not indicate any effect (per Table 2) on P7's behaviour. In each iteration, coder1 also shared the classification of the excerpts of the emerging concepts (antecedents) to be validated by others. These steps ensured that coder1's bias was minimised and coding is reproducible.

4 RESULTS

We have identified 20 antecedents, which are classified into nine categories. First, we define the category, then we introduce the respective antecedents with the description and evidence of *why and how* they influence the dis(confirmatory) behaviours. The definitions of the formed categories are based on the collected data. Table 3 summarises the answers to RQ1 and RQ2. The columns *C* (confirmatory), *D* (disconfirmatory), *B* (both) and *U* (unknown) present for how many participants the respective antecedent influenced as C/D/B/U for their behaviour. The *Total* column presents the total number of participants and the total number of excerpts that provided the evidence for C/D/B/U for the respective antecedent as $x(y)$. It is important to note that x is not a sum of the counts reported in the previous columns because in a few cases the same interview provided multiple evidence. *Why & How (RQ2)* summarises how the antecedent influences the behaviour. The example evidence excerpts presented in this section are revised from grammatical and comprehension perspective because the interviewees were not native English speakers.

4.1 Experience

Experience refers to the knowledge of an individual tester that they acquired by working in either a different, similar or the same project, roles and company, and the application of this knowledge for software testing. The data analysis showed that *experience* mainly leads to disconfirmatory behaviour. However, it also promotes a confirmatory behaviour, which results in improved completeness of a test suite.

4.1.1 Past Experience

Past experience refers to the experience of the participant in general or they did not associate it with any particular project domain. It promotes disconfirmatory behaviour by designing inconsistent test cases. For example, *"You are making the happy path [consistent scenarios] and then you are making some negative scenarios, that are functional or communicated by the customer. And then, you think of the need for extra test cases, which are not written [in specifications], but you should figure them out based on the decision points [of the functionality] or other similar conditions that you have learned from your past experience... Most of such test cases are failure cases [inconsistent] because the happy cases should be [written] in the documents."* - P3. In this evidence *extra test cases* contextually refers to inconsistent test cases. Past experience also results in the occurrence of both behaviours because an experienced tester knows how to approach an SUT, as explained by P7: *"...the more you do testing, the more you get experienced, and the more you know how to approach a product or a system... I can say both [consistent and inconsistent]. I cannot comment about the preference of one over the other"*.

4.1.2 Project Experience

This experience is acquired either by working in the same project over the years or in the same domain, which may not be limited to the Company-ICT. Project experience renders an enhanced perspective on the project due to which disconfirmatory behaviour manifests. This is explained by P10: *"I use my experience in this project because I know the customer, I know the domains in [customer's company name], and it gives me many advantages because I can view the system, when it starts and runs..."*; it helps in designing more *"Exceptional paths [inconsistent tests]"* - P10. This antecedent also increases the completeness of the suite by prompting both behaviours, e.g., P9 stated: *"I joined this project when it started four years ago. As the project scaled up over the years from a little code base, little tests; I learnt the project well all along. Due to that knowledge, I can see [visualise] the end-to-end part of the [domain name]... 90% of the times, I can know, yes this is an exceptional [inconsistent], and yes this is a happy path [consistent], and it is important to test"*.

4.2 Priority

It relates to the priority of a functionality and user stories or OS platform (e.g., Android). Our data informs that priorities are defined by customers, roles higher in a hierarchy to a software engineer (e.g., product owner, project manager, test manager) and it is based on the functionality (e.g., finance). If the higher management or customers are not setting the priority, then testers themselves define them based on their experience, i.e., *past experience* or *project experience*.

4.2.1 High Priority

In the context of automating manual test suites by an automation engineer, higher priority scenarios take precedence. According to the data, testing of consistent scenarios usually have a higher priority. However, if a functionality or particular scenario is a high priority, then it also leads to both behaviours. P6 explains this in the context of automating a manual suite that they, then, automate consistent and inconsistent test cases with equal priority: *"...It should be definitely both [consistent, inconsistent]. When we are talking about the top priority test scenarios or the negative scenarios [inconsistent], it's almost equally important like a happy path [consistent] test scenario."* Our analysis further suggests that high priority testing may not lead to an enhanced coverage; we discuss this in Section 5.

TABLE 2
Data Extraction

Concept	Definition
Antecedent	Of the testing process (e.g., reviews) OR from the environment (project, organisational) OR personal attributes (e.g., experience) that leads to the manifestation of a (dis)confirmatory behaviour.
Antecedent Classification	
Confirmatory	The evidence indicates the promotion or manifestation of a confirmatory behaviour.
Disconfirmatory	The evidence indicates the promotion or manifestation of a disconfirmatory behaviour.
Both	The evidence indicates the manifestation of confirmatory and disconfirmatory behaviours without stating the relative preference of one behaviour over the other.
Unknown	The effect on behaviour is evident but leading to either a confirmatory or disconfirmatory behaviour is not explicitly mentioned.

TABLE 3
RQ1: Antecedents and Evidence, RQ2: Why and How

Antecedent (RQ1)	Why & How (RQ2)	C	D	B	U	Total
	Experience					
Past Experience	Experience in testing enables how to approach the system, and which particular functionality checks to test for.	0	3	1	0	4 (6)
Project Experience	It enables testers to visualise the end-to-end functional flow of a system, and a good learning of the customer domain.	0	5	4	1	10 (28)
	Priority (of a functionality, user stories or platforms)					
High Priority	Consistent test cases have higher priority, but inconsistent test cases also acquire an equal priority in case of high priority scenarios, of a certain functionality.	2	0	2	0	4 (11)
Medium or Low Priority	Testers give either less or no consideration to the designing or execution of inconsistent test cases.	2	0	0	0	2 (3)
	Requirements					
Ambiguous Requirements	Impedes correct and complete test case designing, which results into the design/execution of mostly consistent test cases.	3	1	0	2	6 (14)
Clarifying Requirements	Clarifying ambiguous requirements leads to the designing of both consistent and inconsistent test cases.	0	1	3	2	6 (17)
Incomplete Requirements	Confirmatory because it limits the testers to test only what is minimally specified.	1	0	0	1	2 (2)
	Functionality Retesting					
Production Bug Fix	Testing the fix first is confirmatory that is followed by the testing of relevant inconsistent cases and other consistent test cases.	0	1	3	2	6 (10)
Change Request	Both behaviours occur while designing/executing cases for a change request.	1	0	5	1	7 (8)
Change/Fix Size	Minor change is confirmatory due to its minor impact on the system. Both behaviours occur in case of a major functional change.	3	0	2	0	3 (9)
	Test Suite Reviews					
Internal Party Review	When performed by members of the same project, it is disconfirmatory, and also enhance the completeness of a suite.	1	7	3	2	6 (26)
External Party Review	By customers who define which devices to test, and set priorities. Therefore, only adhering to those priorities is confirmatory.	1	0	0	1	1 (9)
Automated Test Suite Review	Manual testers review to validate conformity with the manual suite. They are not expert in automation to assist with the handling and coverage of inconsistent test cases.	1	0	0	2	3 (9)
	Testing Mode (manual or automated)					
Automated Testing	Confirmatory because it is difficult to automate every inconsistent test case and to handle unexpected results.	2	1	1	0	4 (5)
	Test Execution Feedback					
Detection of Errors	It leads to further testing to find more errors, and sometimes the addition of more inconsistent test cases.	0	5	0	0	5 (7)
Absence of Errors	Disconfirmatory because it leads to rethinking of the test approach, and assessing a test suite from a different perspective.	0	4	0	0	4 (7)
	Time					
Time Pressure	Consistent test cases are prioritised because they ensure the behaviour of the SUT per the documented specifications.	4	1	3	1	9 (27)
No Time Pressure	Leads to more testing e.g., through exploratory testing and the execution of more inconsistent test cases.	0	1	0	2	3 (8)
	Perspective Change (testing from a changed perspective)					
Developer & Tester	Disconfirmatory because testing as a tester, compared to development, changes and broadens perspective for the SUT.	0	1	0	0	1(7)
Complement Testing	A tester executing test cases that were previously executed or designed by another tester promotes disconfirmatory behaviour.	0	2	0	1	3(5)

4.2.2 Medium or Low Priority

If functionality is not high in priority, then it could be a medium or low priority. In this case, inconsistent test cases receive either less or no consideration in designing/execution, which results in a confirmatory behaviour manifestation by a tester. P9 explained it as: *"If a function is important, all the happy path [consistent] and the exceptionals [inconsistent] are also important. But, some functions may not be very important, and we can skip the exceptional scenario for not important functions"* - P9.

4.3 Requirements

This category refers to the documents that serve as requirements specifications for testers to prepare test cases. It includes technical documents, business rules, functional documents, high-level design documents and low-level design documents.

4.3.1 Ambiguous Requirements

Ambiguous requirements are such requirements that are either not well defined or are difficult to understand by the testers. According to P2, such requirements affect the activity of preparing test cases because testers have to ask for clarifications; *"...these requirements might be not very clear, sometimes you might need to ask more questions about the documents, to be able to make all your test cases clear and comprehensive enough, to be able to test the system"* - P2. Other participants' data indicates that ambiguous requirements promote confirmatory behaviour. Testers design more consistent test cases because it helps them in understanding the requirements well to design inconsistent test cases, afterwards. Otherwise, they only design and execute consistent test cases based on their own understanding. For example, *"If I do not understand what's going on [in requirements], then I'm not able to write test cases... I code [design] happy [consistent test cases] just because I'm not clear with what they expect me to do [test]. And, I'm not sure what system does, so I go with happy path and if products do not crash, then I say it's okay"* - P12.

4.3.2 Clarifying Requirements

Clarifying requirements is an activity that is performed by testers to clarify ambiguous requirements to improve the testing of a functionality. Testers clarify the requirements with customers, product owners, developers or project managers. It usually leads to the completeness of a test suite when testers manifest both behaviours. P1 states this as: *"I don't know [understand] all of them [the requirements]. I will exchange my comments with customers, whether I am understanding them right, or maybe it's not required [a particular functionality], it's not end-user's behaviour. I will give comments about all of them... Yes, both [consistent and inconsistent], I will check all of them"*.

4.3.3 Incomplete Requirements

This antecedent is different from the above antecedents because it refers to minimal requirements. In other words, requirements may be ambiguous but may not lack details on the required functionality to be tested. For example: *"If no information [is available] about the task. For example, they [authority figure for preparing the documents] wrote only a single sentence about a problem's fix on the production, and developers fix it. It's sometimes difficult for the tester to understand [the functional fix], what did he [the developer] do, and what was the real problem"* - P10. The incomplete specifications, in case of a production fix, make it difficult for a tester to perform proper testing because they lack details on the functionality. It promotes confirmatory behaviour because testers, test per the minimal information that limits testing inconsistent scenarios. As further explained by P10: *"it will affect, what I don't know [the requirements], so it affects my test cases... Exceptional or failure ones [inconsistent]. Because I don't know the details. Only focus on the happy paths [consistent], maybe I miss [testing] something"*.

4.4 Functionality Retesting

Retesting refers to retesting a module or functionality after its re-implementation, in case of a reported production bug or a functional change request. In addition to retesting of a particular fix or change, it is also done for the relevant impacted functionalities of the SUT.

4.4.1 Production Bug Fix

The data analysis showed that both behaviours occur due to retesting a fix of the production bug. After validating the fixed scenario, the tester begins to test the inconsistent scenarios of the module, which is followed by the testing of consistent scenarios. For example, P10 stated: *"Firstly failures [that failed], and then check the happy paths... I also ask the developer, "which code did you change and which cases does it affect?". First we talk, then I check the failure one, and [then] check the success path"*. Per this evidence, the tester firstly validates particularly a fixed scenario. It is a confirmatory behaviour because they are confirming the communicated (serving as a requirement) functional flow. Then, a disconfirmatory behaviour when they validate the other relevant inconsistent scenarios, which is followed by the testing of other confirmatory test cases.

4.4.2 Change Request

Retesting a module, in case of a change request mostly leads to both behaviours. For example in the context of automation testing, P8 stated: *"We should delete some methods, and we should have some other control points, and add some other modules or functions to the automation framework... Both. Happy [consistent] paths and exceptional [inconsistent] and failure scenarios"*. On further enquiry, the participant explained that firstly they prefer to test consistent scenarios.

4.4.3 Change/Fix Size

The size of the implemented change or a bug-fix also influences the behaviour of testers. In case of a minor change, testing is confirmatory and limited to a particular functionality. For a major change, the manifestation of both behaviours was reported. A type of a major and minor change is elaborated by P5 as: *"for example some text box or button is not in the right place on graphic user interface. Either it is there or is not visible. So, I just test this because it's a makeup thing, just an interface issue. It's not a major big problem. But, if I cannot make any stock or product transfers, i.e., main function is not working at all, of course, that means that all product transfer*

function will be tested from the top to down". However, time availability also plays a role in retesting: *"if you don't have much time; e.g., if you have a small change, it's not affecting all the release, all the software outcome, if it doesn't affect every part of your platform, you can just run a quick happy path [consistent] test cases"* - P2.

4.5 Test Suite Reviews

It is the review of test suites that are designed by testers, prior to suite executions, to ensure the completeness of the suite with respect to the SUT. The antecedents of this category indicate two types of reviewers, which promote different kinds of behaviours among testers based on their review-feedback.

4.5.1 Review By Internal Party

These reviews are conducted by the roles who are employees of the Company-ICT. They may be part of the same project or team, i.e., project manager, solution architect, product owner, team lead, development lead, test expert or fellow testers, or testers from other projects. The reviews from members of the same project/team mostly promote a disconfirmatory behaviour by recommending to accommodate more inconsistent test cases. P11, who also reviews others' test suites, stated: *"Generally they forget exceptionals [inconsistent] scenarios because they argue that it works. But, I check and [fore]see other different bugs. And generally I suggest, "You can write some exceptional scenarios; [e.g.,] sometimes bad things [situations], sometimes field checking; it's important"* - P11. According to P3, reviews enhance the completeness of the suite: *"When I am adding, most probably, you are not adding the happy [consistent] cases. When you are sending it for review, a very small part, maybe five per cent that they are arguing or asking for an extra [test cases]"*. On enquiring the type of *'extra'* cases, P3 replied: *"It's changeable because they are giving review, which you forgot about [test cases]... Both [consistent and inconsistent]"* - P3. The data analysis also shows that reviews performed by testers from other projects are confirmatory because they are not knowledgeable about the functionality. It limits their perspective that could promote disconfirmatory behaviour.

4.5.2 Review By External Party

External reviews are performed by customers. The purpose is to get their feedback, if the suite meets their expectations, to continue with the test execution. Per P2: *"We have shared this with customer, if these test cases, test suites meet their expectations to be able to test the system... We test on mobile platforms, e.g., iPhone, iPad, Android tab, Android phone. So customer can say, it is enough for us to execute the tests only one Android device, and only one iOS device. This is enough. So, "Continue your tests on the set top box, which is more important for us." They can say this. So, we have to consider this"*. The review from customer influenced the coverage, which in this case is limiting testing to certain devices. Additionally, the feedback also defined priorities for testing. It is confirmatory because the tester is confining the testing only to the customer's feedback, per the available evidence.

4.5.3 Automated Test Suite Review

Reviews of automated test suites are internal reviews. However, it is different from internal and external reviews because those are performed only for manual test suites. Contrary to the range of roles involved for manual suites, automated test suites are reviewed only by manual testers. The major reason for this is, automated suites development is based on manual suites. P6 stated this as: *"For the main sources are manual test scenarios. We are expected to automate the manual test scenarios as it is, the same steps, the same verification points, the same databases... we have only the manual tests and they just want us to simulate it"*. The manual testers usually assess and compare the functional flow of the automated tests with manual tests. Therefore, the quality (the level of completeness) of manual suites gets transferred to the automated suites, as P3 stated: *"you are simulating the manual testing, so you should take a proof review on the manual testing; It's OK or not"*. P6 explained: *"the common observations they [manual testers] are giving are observation on the happy path [consistent] test scenarios. But, the exceptional [inconsistent], there are some experienced test engineers, giving some feedback about the negative [inconsistent] scenarios, but this is less, maybe one in a ten"* - P6. This evidence cannot be considered as *Both* because the frequency of feedback that can lead to the addition of inconsistent test cases is considerably low.

4.6 Testing Mode

This category refers to the mode of testing, i.e., manual testing or automated testing.

Automated Testing

Automated testing is the testing performed in an automated way using tools, e.g., Visual Studio, Selenium. Test automation engineers develop automation scripts that run the tests in an automated way. In comparison to manual testing, automated testing leads to confirmatory behaviour. It is because of the difficulty to code inconsistent test cases and automation tool's limitations in this regard. For example, P3: *"Automation is more [about] happy cases, you can say that. Of course, it can handle negative [inconsistent] test cases, but with manual testing, negative cases or unexpected times or results can be handled better. With automation test cases it's more difficult to handle unexpected results"*. Also, P8 complemented to this; *"... from manual testing opinion you should test the whole thing. Because you are a user on the computer, you are using that computer with your hands, with your mouse. You can do anything in that time. But in automation testing, you should write a code. This is not the natural way. This is about the priority of testing, I think. The automation testing mainly focuses the happy paths"*. According to P2, a tester can benefit from the confirmatory nature of automated testing due to its fast execution time. *"If you automate 200 cases... you can run them e.g., in one hour or two hours. Rather than spending a day or two man-days. So, this shows you a general outcome, result of this. And you can review it. You can say, "Okay let's now concentrate on these failure [inconsistent] cases because we haven't automated [testing of] these cases and these features. Let's focus on them." This gives you a good time to focus on other features, areas and failure cases"* - P2.

4.7 Test Execution Feedback

This category refers to the effect of the results of a test suite execution especially when a module is tested the first time. The data analysis has revealed that detection of errors and also an absence of errors lead to a disconfirmatory behaviour by testers.

4.7.1 Detection of Errors

This antecedent refers to the situation when a test case failed due to the presence of error. It is disconfirmatory because it leads testers to find more errors in the SUT by performing further detailed investigations. In case of P9 it leads to exploratory testing, i.e., performing more manual testing for finding errors, hence, a disconfirmatory behaviour: *"When I see a bug, I first open a trouble report... And after that I think, [if] there is a bug, maybe other scenarios are also troubled. And also, I do free tests [exploratory testing] at that part, at that time maybe... if there is one bug there must, there can be other bugs. And, I investigate that part, and sometimes I get [detect] other bugs, sometimes not"* - P9. It also leads to the addition of more test cases, as P5 stated: *"if I find a new important bug, I go deeper, and I also won't let the test cases [go] unseen. I still run the cases, and if I find some exceptional cases that I couldn't consider before, I know they're exceptionals, I go deeper, too... Maybe I didn't consider [it] before [test cases], and it's also not written in the requirements, So I write it down"*. This antecedent's influence on the behaviour is observed only for manual testers, not for automation test engineers - discussed in Section 5.

4.7.2 Absence of Errors

This is a situation when all test cases of a suite pass, i.e., no error is detected by the suite for the SUT. This promotes a disconfirmatory behaviour among testers because it makes them curious over the situation and to rethink of their test approach. Per P10: *"I always think, I'm doing something wrong. "How [could] they develop with no bug?"*. P2 explains this situation as: *"If you can't find some issues with your test set, there might be issues in your test set approach. So you should be able to consider error cases [inconsistent], failure cases [inconsistent], what's going on. Go over your documents, test sets, and then detail some of them, change your mindset, how you created them"*. Hence, this situation also prompts testers to force the system from a different perspective to reveal its errors. In case of automation testing, testers execute a few test cases manually to reassure a 100% pass result. *"Green is kind of a very relieving colour, and when you see green all over, you feel very happy. Of course, we are investigating, we are just executing manually a couple of test scenarios. Let's see [if] it really passed all the test scenarios"* - P6. It is a disconfirmatory behaviour because it led the tester to investigative more rather than being contented by the test results of automated suite.

4.8 Time

This refers to the available time for two main testing activities: test suite designing and test suite execution.

4.8.1 Time Pressure

It is an insufficient time availability from testers' perspective for performing testing in the situations when they do not arrange overtime. The data analysis shows, in this situation, most testers manifest confirmatory behaviour because they prioritise to validate that SUT accomplishes the specified functionality. Inconsistent test cases fall second because they validate implicit functionality, i.e., not explicitly mentioned in the specifications. P10 stated: *"I want to test more but I have a limited time. I only check the happy path [consistent] or, one or two exceptional [inconsistent] test cases. But I think, I should test more and [also] check the other exceptional ones. But I don't have time, and should start [testing] the other project. So, it limits my execution, I think. Exceptional test case execution"*. Those who were observed to manifest both behaviours, for most of them a confirmatory behaviour occurred prior to a disconfirmatory behaviour. For example, *"If I have really a short time, really short time, of course, firstly I need to see the system is working correctly, the happy path [consistent]. Whether the happy path passes right or wrong. I mean, then exceptional cases [inconsistent] of course. But I have to tell you that happy cases don't take that long time, just pass away"* - P5. Testers who firstly manifest a disconfirmatory behaviour, they compromise on the testing of consistent scenarios. *"So for urgency [time shortage], I first start with exceptional [inconsistent] scenarios. And for urgency sometimes, you make exploratory tests, based on our experience of the product... If it's enough urgent, you sometimes, trust the development team that they should have developed these according to requirement"* - P7.

4.8.2 No Time Pressure

No time pressure refers to two situations: 1) when testers are finished before the deadline, or 2) testers have enough time to perform testing, i.e., without doing overtime. According to P2: *"You should make enough time to run even different tests. Sometimes, we have free time and we don't base it on any test set [designed tests]. We just start testing a mixture of functional and non-functional tests"*. In this case, the tester is performing exploratory testing. The influence on the behaviour is not known because it is not clear, whether they execute consistent test cases or inconsistent ones. However, it definitely leads to the execution of more test cases. However, P9's behaviour is disconfirmatory in this regards: *"... if I have enough time, I also execute free tests... not related with any [designed] test cases. Maybe there are [exist] test cases, but I do not know... I also do [test] the exceptional [inconsistent] cases. For example, in this example [case], maybe more of them will change the status [of the feature] four or five times, but when I test, I change it [the case]"*. P9 also referred to the execution of exploratory testing that consists of inconsistent tests. The test case, in this evidence, is validating the status' feature for the situation for which a test case may not already exist.

4.9 Perspective Change

The change in perspective occurs either due to a change of role or testing the functionality that was previously tested by another tester. The antecedents of this category promote disconfirmatory behaviour because of the changed perspective.

4.9.1 Developer and Tester

A software engineer working in two roles, as a developer and tester in the same project, also influence their behaviour. P1 explained this as: *"when I am a developer, I just focus on happy [consistent] paths, maybe one risky [inconsistent] case but mainly happy path to check it out if it is okay. But when I am a functional tester, I will see every risky point. If I am an integration tester, I will force every possible error from the integration part because it's the most risky thing in our environment. So it changes my approach"*. After a confirmatory behaviour in testing as a developer, testing as tester changes and broadens their perspective of the SUT, which leads to a disconfirmatory behaviour. However, P1 related the reason for disconfirmatory behaviour in testing to their *past experience*.

4.9.2 Complement Testing

This antecedent refers to two situations: 1) a tester executing test cases, designed by another tester, and 2) a tester, testing the functionality that was tested by another tester in the previous test cycle of the same release. P12 explained the first situation as: *"We do not actually check the entire cases because our test lead separates the things that we do. So we are seven people and two or three of them write the test cases and the other three or four, run the cases. And if we find something that was not included in those cases, we add it"*. On enquiring further, they mentioned that the missing cases are usually inconsistent test cases. Hence, the execution of the suite that was designed by another tester prompted a disconfirmatory behaviour because it enabled a different perspective to test the same functionality. This antecedent also leverages improved defect detection, e.g., P4 stated: *"I run the [functionality-1 name], [functionality-2 name], [functionality-3 name], for example. Other LSV [system testing] cycles, my other friends run [functionality-2 name], [functionality-1 name] and [functionality-3 name]. If I miss something, miss a failure, miss defects, maybe she finds it. Therefore, we have little defects"*.

5 Discussion

We first discuss the classification of the identified antecedents and present the integrative diagram, followed by their comparison with the antecedents from the existing literature. The section also presents the implications for research and practice. Finally, the threats to validity are discussed.

5.1 Classification of Antecedents

Based on the results, we can classify the antecedents from three aspects: confirmatory, disconfirmatory, and *both*, that represents the test suite completeness perspective. The categories: *test execution feedback* and *perspective change*, and the antecedents: *no time pressure* and *past experience* are disconfirmatory. The antecedents: *medium or low priority*, *incomplete requirements*, *external party review*, *automated test suite review* and *automated testing* are confirmatory antecedents. *Ambiguous requirements* can also be classified as a confirmatory antecedent. The rest of the antecedents, in addition to providing the evidence for confirmatory or disconfirmatory behaviour, also provide evidence for the manifestation of both behaviours. It is important to note that *both* may not suggest a complete test suite, albeit an improved suite. For example, the antecedents: *change request* and *clarifying requirements* mainly lead to improved completeness of a test suite. *Production bug fix* leads to an increased execution of the test suite because it is in the context of retesting, as indicated by the antecedent's category. For the two antecedents, *high priority* and *time pressure*, *both* does not suggest completeness of the test suite in terms of design, and also a complete execution of a test suite. It is detailed later in this section. The antecedents: *project experience* and *internal party review*, in addition to the promotion of disconfirmatory behaviour, also improve the completeness of the suite. *Change/fix size* is a special case because, for major change/fix, it is both the behaviours, otherwise it is confirmatory.

Despite the proposed classification of the identified antecedents, our data suggest that exclusive classification of these may not be practical. If an antecedent leads to confirmatory behaviour for some testers, it may also lead to disconfirmatory behaviour for other testers, which could be pertained to certain factors, e.g., personality elements. This issue and other particular aspects related to the identified antecedents are detailed further.

General and Specific Behaviours: The data informs that the general behaviour of testers is to firstly manifest a confirmatory behaviour, i.e., designing of consistent test cases and then a disconfirmatory behaviour, which is designing of inconsistent test cases. It happens because they identify inconsistent test cases based on consistent test cases. The automation engineers reported the same behaviour sequence when they automate the manual suites, though they only simulate the manual flow. In addition to designing the test suites, test execution also follows the same course, i.e., first confirmatory and then disconfirmatory.

A few participants manifest the opposite sequence of behaviours - specific behaviour. They manifest disconfirmatory attitude prior to confirmatory attitude, which is pertained either to their *experience*, particular nature or assumptions (developers must have rightly implemented the consistent scenarios). For example, in Table 3, the evidence of specific behaviour can be seen for *ambiguous requirements* and *time pressure*, though they are confirmatory antecedents. P1 associated this with their experience. Also, this pertains to tester's nature; in the context of ambiguous requirements, they stated: *"It makes me check risky scenarios... Because I think like this, if it is difficult it might be more risky... so I need to see them first"* - P1.

If a test suite designing or execution is complete in terms of consistent and inconsistent test cases, then manifesting one behaviour before the other is not an apprehension. However, if completion of one type of test cases is compromised (due to the antecedents), then the behaviours may lead to adverse effects on software quality, as already explained in Section 2.

Functionality Retesting: For testing change requests or production bug fix, the same sequence of general and specific behaviours occurs. *Project experience* also influences the preferred behaviour manifestation by the participants, i.e., confirmatory or disconfirmatory.

High Priority Testing: The testing with inconsistent test cases along the testing with consistent test cases (manifestation of disconfirmatory behaviour leading to *Both* in Table

3), for higher priority functionalities or scenarios, does not imply improved completeness of a suite design/execution. Since, the testing is still limited to the higher priority items.

Requirements and Agile Software Development: The participants of this study belonged to projects that apply agile software development method Scrum. Hence, the antecedents of the *requirements* category (*ambiguous*, *incomplete* requirements) may be confined to this software development method. In other words, the emergence of this category suggests a high dependency on Scrum. P2 stated: *"The actual reason, why in this project agile scrum is used, customer sometimes might give you less details, less detailed requirements. So, this causes some issues. Also, you might forget to get [obtain] enough detailed requirements. So, this is a very normal situation. It happens in all projects. That's why agile scrum is used in this project"*. Paetsch et al. stated that agile software development is more adaptive to frequent changes, and is more reliant on direct collaboration instead of documentation oriented processes [36]. As a result, agile is more "code-oriented" and less "document-centric" [36]. Therefore, the requirements to be implemented in the following sprint might not be comprehensive for testers to design complete test suites.

Clarifying Requirements: This antecedent of *Requirements* category promotes *Both* behaviours but it may not always be possible to clarify requirements, e.g., in case of time pressure.

Detection of Errors and Automated Testing: Detection of errors does not promote any kind of behaviour among test automation engineers. The reason for this could be that automated testing, compared to manual testing, do not require an active involvement of a tester during the test suite execution. Once the complete script is run, the results are generated, which are then investigated by the automation tester. The dependency of automated suites on the manual suites may also be a reason for this observation, i.e., the automation testers may find/receive a complete manual test suite to automate.

We could not observe the level of automation as an antecedent to the behaviours of testers performing automated-testing. According to the results, automated suites are developed based on manual suites. Therefore, the modules that are not fully automated, are possibly manually tested. Nonetheless, a possible effect of (the level of) automation is mentioned in Section 4.6, i.e., the fast execution time of automated testing creates time for the (manual) execution of difficult-to-automate inconsistent test cases and other modules that could not be automated. This may promote disconfirmatory behaviour among manual testers.

The analysis also could not support the effect of testing-tools on the behaviour of testers. The participants reported using the tools for maintaining test suites, designing and execution of test suites, test cases statuses, assignments of test cases to others (e.g., to developers for fixing), progress tracking, generating test reports and having a shared platform. These support the testing process, either manual or automated. This may not affect a tester's (dis)confirmatory behaviour except, for example, the stage of testing (designing or execution) or other reported antecedents (Section 4) inclusive of the general and specific behaviours. For example, P1 reported that the tool they use does not affect their (dis)confirmatory behaviour.

Time Pressure: The data analysis also suggests that *time pressure* leads to *high priority* testing, whether a manual or automated testing. It is also found to affect the practice of exploratory testing that some testers perform, e.g., in the case of *detection of errors.* Moreover, the participants have reported applying *experience* under time pressure for performing effective testing. Based on the experience, they choose the execution of test scenarios that can be either consistent, inconsistent or both. As P2 explained: *"when time is pushing and both of things [time and previous experience]... You can choose which parts to test. This is like an instinct from your previous experiences and history of the project. You know things [functionalities] and then you find the best approach..."*. Furthermore, the participants attributed limited time availability to Agile practices. A study by Linß et al. found ten antecedents to, and five consequences of time pressure, by analysing time pressure in software projects that apply Scrum [37]. It is evidence that time pressure is intrinsic to the agile development method - scrum.

Figure 2 presents the integrative diagram of the formed categories based on the identified antecedents. The antecedents are separated with a semicolon (;) inside a category box, followed with their classification, e.g., '*D*: Detection of Errors'. The arrows depict relations among the categories, i.e., how one category is influencing the other, e.g., customers define priorities for platform or functionalities (*priority*) when they perform reviews (*external party reviews*). This is indicated by an arrow sign from the *Test Suite Reviews* category to the *Priority* category. The relations between the categories are based on the relations between the antecedents of those categories. These relations are a figurative depiction of the narrative in the results and discussion section. In the figure, the *time pressure* can be seen as impacting other categories by promoting confirmation bias and limiting the completeness of test suite design or execution. The *time pressure* is also diminishing the possibility of exploratory testing, thus decreasing the disconfirmatory effects of the antecedents of *test execution feedback*. In the holistic perspective, the *experience* has emerged as a decisive category for the specific or general behaviour manifestation for other categories, and a contributor to the *priority* category.

Conclusively, confirmation bias is manifested due to the confirmatory antecedents because consistent test cases are designed/executed relatively more than the inconsistent test cases. The antecedents that lead to the disconfirmatory behaviour, and also to a complete test suite (design or execution), suggest the possible mitigation of confirmation bias. Fischoff mentions five levels of debiasing[7] interventions: a) warning about the possible bias, b) describing the direction of the typically observed bias, c) personalised feedback, d) training for cognitive mastery and e) debias the task instead of the person [10], [39]. *a*, *b* and *c* are seldom effective, and *d* is expensive, hence, Fischoff proposed *e* [10], [39]. In this perspective, a few of our disconfirmatory antecedents are task/practice-oriented, e.g., *complement testing* that may debias confirmation bias. The antecedents that mostly pro-

7. It refers to preventing or alleviating the effects of cognitive biases [10], [38].

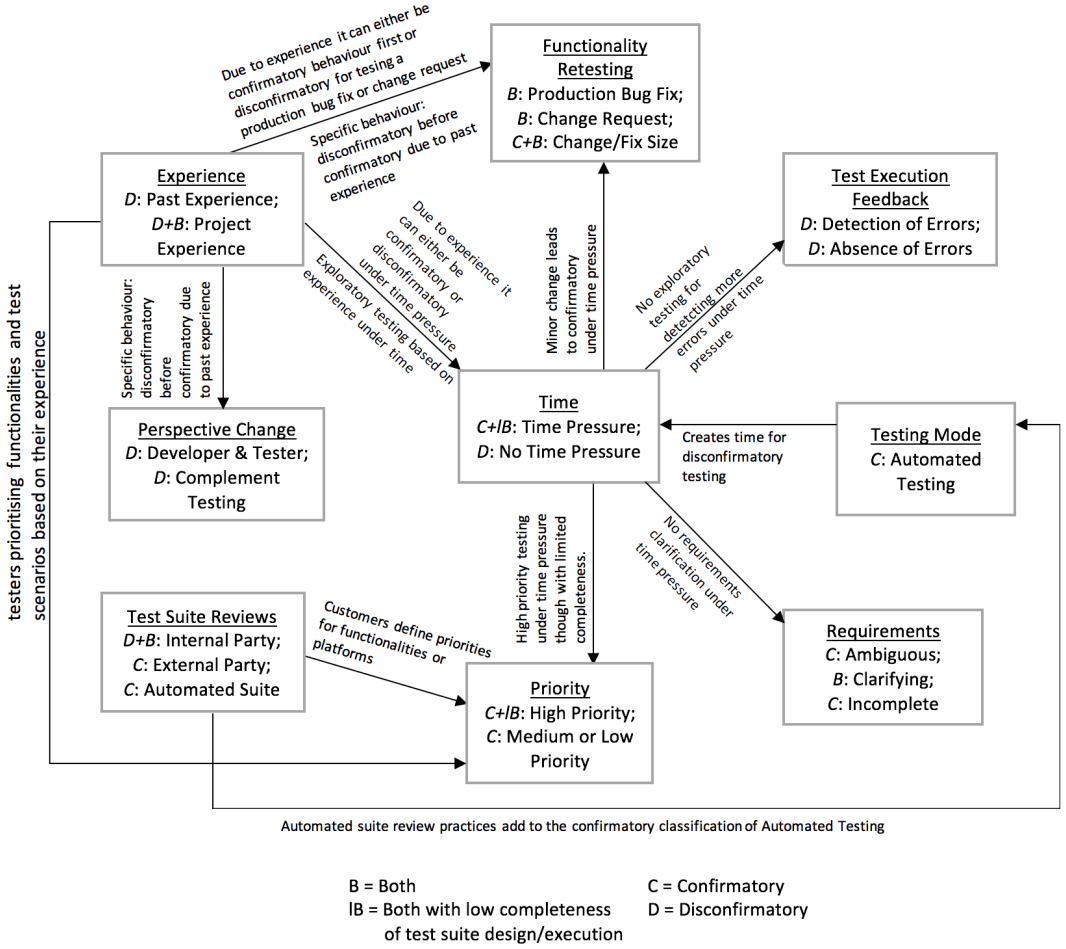

Fig. 2. Integrative Diagram depicting the relationships of the identified Categories and Antecedents to (Dis)Confirmatory Behaviour

mote both behaviours for testers, e.g., *change request*, lead to improved completeness of a test suite, if not interrupted by *time pressure*. When time pressure occurs, it affects the completeness of either consistent test scenarios or inconsistent test scenarios, which is respective to the general or specific behaviour of a tester. The specific behaviour (first disconfirmatory) in such a situation although could mitigate confirmation bias but may not assure a defect-free SUT. Since, the consistent test cases are not executed by the tester, that may fail.

5.2 Comparison with existing Literature

Table 4 presents a comparison of the identified antecedents to the antecedents found in the existing literature. Five out of 12 antecedents of the existing literature are comparable to seven of the antecedents of our study. However, the existing literature does not empirically support the effect of two of the five antecedents on confirmation bias. Our study identified 13 new antecedents compared to the existing literature. There are seven antecedents that the existing literature investigated, but our study could not identify them. However, the existing literature does not empirically support the effect of four of these 7 antecedents on confirmation bias. Table 4 uses different symbols ('E', #) for presenting the effect of antecedents because the existing literature uses different assessment methods for measuring confirmation bias. Additionally, our study is a qualitative study and the existing literature are quantitative studies. No effect symbol ('E', #) before the antecedent indicates that the existing literature could not experimentally observe it to affect confirmation bias.

The category *experience* and the related antecedents of the existing literature are similar because they all point towards the possible mitigation of confirmation bias due to experience of the testers. The antecedent, *completeness of specifications*, which led to the lower levels of confirmation bias in the studies of Leventhal et al. and Teasley et al., is relatable to A6 of our study [6], [7]. Since, the results of A6 may lead to the complete elaboration of all the required and non-required functional behaviour of the SUT. Leventhal et al. and Teasley et al. defined three levels of specifications [6], [7]. The first and second levels, minimal and positive only specifications are similar to A7 because they all suggest a manifestation of confirmation bias. The effect of the antecedent, *error feedback* was investigated in the context of the presence of errors vs. absence of errors by Leventhal et al., which remained inconclusive [6]. Contrary to the hypothesised effect of *error feedback* by Leventhal et al. that absence of errors may not decrease confirmation bias levels [6], our study suggests that not finding any error (*absence of errors*) also promote the disconfirmatory or code-breaking behaviour among testers. The disconfir-

matory behaviour manifestation in our study may be attributed to the extensive industrial testing experience of the participants. Whereas, Leventhal et al. employed graduate students to represent advanced testers, whose maximum professional experience did not exceed over a year as an intern-programmer [6]. Our study showed that *time pressure* is a confirmatory antecedent as well as an antecedent that is ineffective for testing from the test completeness perspective. However, Salman et al. could not find it as a promoting factor for confirmation bias in their experimental study [3].

The participants of our study did not refer to their *logical reasoning skills* (acquired through their education) or *educational background and levels* for their (dis)confirmatory behaviours, in contrast to the evidence shown by Calikli and Bener [5], [8]. The antecedent, *job title* cannot be compared with any antecedent of our study because we did not segregate based on roles. We considered all roles performing core testing oriented activities, e.g., test suite designing and test suite executions, as testers. Additionally, all the participants were practitioners, therefore comparison with *researcher* aspect is impossible. *Company culture*, *company size* and *development methods* are also not directly comparable with any of our antecedents because our data collection was limited to one company only and none of our participants was solely a developer. It is important to mention that the antecedents with no effect on confirmation bias levels, in the existing literature, are due to not statistically significant results. The discussion on the observed effect sizes of those antecedents, which may imply a possible effect, is beyond the scope of this study.

5.3 Implications for Research and Practice

For research, we propose a multi-case study to explore, whether the antecedents found in this study also hold in other settings because the results of our study are confined to the testers of one company only. A cross-case analysis would also aid towards finding the influence of the antecedents that are particular to the companies, e.g., organisational culture, development methods, company size, as per quantitatively investigated by Calikli and Bener [5], [8]. More studies applying grounded theory using multiple data collection methods, for exploring the same phenomenon, may reveal new antecedents with more intense evidence. According to our results, ambiguous and incomplete requirements promote a disconfirmatory behaviour, however, under time pressure this leads to a confirmatory behaviour manifestation. More studies are needed on how to improve the requirement specifications that may deteriorate software quality especially in the context of Agile that also constraints time [37]. Yet, the manifestation of confirmation bias in the case of complete requirements is not detrimental for software quality because a tester is then validating all the specified required and not required behaviours of the SUT [3]. In the context of Agile, whether to improve the requirements or to devise solutions to manoeuvre the possibly limited time, is a question that needs scientific attention.

Experimental studies and experimental replications would help strengthen the evidence quantitatively of the identified antecedents. Experimental studies may also help find the relative importance of the identified disconfirmatory antecedents for effective testing, e.g., presence of errors vs. absence of errors, and how influential is the role of experience (general experience in testing vs. project/domain experience) in this comparison.

We recommend the following *to practitioners* :

Test Suite Reviews: It is important to implement internal test suite review practices if they are not already in place. It is critical that manual test suites are reviewed by the team members of the same project. The same project members are better able to promote disconfirmatory behaviour and also enhance the test suite completeness because they are knowledgeable on the project or domain of the SUT. Despite the review by customers, of the manual suites, internal reviews should still be conducted because customers may focus only on defining the priorities of the functionality rather than promoting a disconfirmatory behaviour. Once the quality of manual suites is assured - a suite that is disconfirmatory and improved in completeness, the dependence of automated suites on manual suites may not be deteriorating for the quality of testing. However, expert test automation engineers should review the automated suites to help less-experienced automation engineers to develop complex test cases especially the inconsistent ones. This could improve the coverage of inconsistent test cases alongside the learning and manifestation of disconfirmatory behaviour by test automation engineers.

The recommended practice of test suite reviews may also be interrupted by time pressure. In such a case, testers with *project (specific) experience* or practice of *complement testing* may cover for skipping test suite reviews. Project experienced testers could be able to achieve possible completeness in designing/executing a test suite(s). The practice of complement testing, i.e., test case designing and execution by two different testers may accommodate more/missing inconsistent test cases to the suite. Thus, ensuring improved completeness for test suite execution.

Experience and Test Execution Feedback: Modules developed by experienced testers may appear less defective or defect-free to inexperienced testers. These modules should be tested by experienced testers because the apparent *absence of errors* may prompt more code-breaking (disconfirmatory) behaviour among them compared to inexperienced testers. This may lead to enhanced coverage of inconsistent test cases for the module, which may also reveal errors.

Time and Complete Test Execution: In order to increase test suite execution in terms of (in)consistent test cases, under time pressure, manual test engineers should work in collaboration with automation test engineers. For example, automation engineers run the automated test cases and manual testers run the cases that could not be automated for the SUT. This collaboration may make efficient use of the limited available time with an improved test suite execution. The collaboration may also support other situations that may suffer due to time pressure, e.g., complete execution of inconsistent (manifestation of disconfirmatory behaviour) and not high priority test cases. Automation can be run for not high priority test cases, and manual testers can validate the rest of the functionalities and test cases. Functionality retesting may also benefit from the collaboration in the same manner for time-pressured situations.

TABLE 4
Comparison with Existing Literature

No.	Our Study	Existing Literature
	Experience	↓ Expertise level [6], [7]; ↓ Experience and activeness in testing and development [5], [8]
A1	D: Past Experience	
A2	D+B: Project Experience	
	Priority	
A3	C+lB: High Priority	-
A4	C: Medium or Low Priority	-
	Requirements	
A5	C: Ambiguous Requirements	-
A6	B: Clarifying Requirements	↓ Completeness of Specifications [6], [7]
A7	C: Incomplete Requirements	
	Functionality Retesting	
A8	B: Production Bug Fix	-
A9	B: Change Request	-
A10	C+B: Change/Fix Size	-
	Test Suite Reviews	
A11	D+B: Internal Party Review	-
A12	C: External Party Review	-
A13	C: Automated Test Suite Review	-
	Testing Mode	
A14	C: Automated Testing	-
	Test Execution Feedback	Error Feedback [6]
A15	D: Detection of Errors	
A16	D: Absence of Errors	
	Time	
A17	C+lB: Time Pressure	Time Pressure [3]
A18	D: No Time Pressure	-
	Perspective Change	
A19	D: Developer & Tester	-
A20	D: Complement Testing	-
	-	E: Company Culture (of different geographic regions) [23]
	-	↓ Logical Reasoning Skills [5], [8]
	-	↓ Job Titles (researchers vs. tester, developer, analyst) [5], [8]
	-	Development Methods (e.g., incremental, agile and TDD) [5], [8]; TDD vs. TLD [9]
	-	Company size (large, small and medium enterprises) [5], [8]
	-	Educational Background (undergraduate) [5], [8]
	-	Educational Level (bachelor's vs. master's) [5], [8]

Key: E = effect on confirmation bias ↓ = decrease level of confirmation bias

5.4 Evaluating the Grounded Theory

We evaluate our grounded theory presented in Figure 2 according to the Glaserian evaluation criteria [32], [40].

One aspect to evaluate ***fit*** of the theory is its ability to explain the realities of the studied phenomenon as per viewed by the participants [40]. We shared the generated theory with the participants of our study. The participants found that the identified antecedents and their relationships represent their testing experience, as per said by P6: *"factors [antecedents] are covering my testing experience"*. The respondents also mentioned that the theory also explains the effects of the antecedents on their testing behaviour, especially regarding the disconfirmatory antecedents. This relates to the ***work*** criterion of the evaluation [32]. In our case, ***relevance*** relates to the theory's appeal for practitioners [41]. We achieved it based on the feedback of the participants, as they agreed with the identified antecedents and their interrelationships in comparison with their testing experience. The last criterion is ***modifiability***, which suggests that the theory is flexible to accommodate variations proposed by new data [32], [40]. We were able to modify our theory as we progressed with the analyses of data. Modifiability continued to appear at two points, first, during the classification of the antecedent as (dis)confirmatory and *both*. Second, during the analyses of the relationships among the emerging categories and concepts (antecedents) because of our additional, *why & how* focus of the analyses.

5.5 Threats to Validity

This section elaborates on the threats to validity of our study.

Transcription of the interviews was affected by the accent of the interviewees because they were non-native English speakers. Therefore, we sent summaries of the transcribed content to the respective interviewees for a confir-

mation on the collected data. However, we received only one response that confirmed the content, other participants did not respond. The interviews conducted without video may not have provided lower quality data because there is not enough empirical evidence to support this possible threat to the data quality [42]. The pilot interview ensured that we collect the right data during the actual interviews. The percentage of disregarded data (Section 3.3) is minor, we do not believe that it could have caused major threats to our results. We achieved code saturation while coding our data. Code saturation is achieved when code book stabilises, i.e., the data do not suggest any further issues (codes), which in our case are the antecedents [43]. According to Hennink et al., it is possible to achieve code saturation by as few as nine interviews [43]. Our participants were chosen based on a common criterion of experience of software testing; this introduces homogeneity in our sample [44]. Therefore, twelve interviews have been sufficient to achieve (code) saturation when the study's objective is to describe the behaviour of a comparatively homogeneous group [44].

In order to mitigate the potential threat of researcher bias while forming an initial list of the terms that indicated dis(confirmatory) behaviour on the tester's (participant's) end, we also coded the evidence that explained participants' understanding of the terms. The initial list of the terms could be a source of bias for the other coders while performing the pilot coding - Section 3.3. However, the low agreement levels between the coders of the pilot coding, do not suggest such a possibility. According to Urquhart, a researcher applying GT should not have *"preconceived theoretical ideas before starting the research"* [31, p. 16]. In our opinion, familiarity with the relevant literature beforehand has not compromised this characteristic of GT because only limited literature is available. Our study is the first that has explored the *why* and *how* aspect of confirmation bias in software testing. Additionally, the identification of 13 new antecedents and absence of seven existing antecedents from our generated theory (Section 5.2) suggest less influence of any preconceived ideas. We acknowledge possible threats to our theory for not performing theoretical sampling because it was not practically possible. Theoretical sampling enables to address the gaps in the emerging theory [31], [32]. It also helps to increase the scope of the theory by sampling other substantive areas [31]. According to Eisenhardt and Graebner, theoretical sampling refers to the selection of cases that are specifically appropriate for "illuminating and extending relationships and logic among constructs [45, p. 27]". From this aspect, theoretical sampling was implicitly applied from the beginning when our contact person sampled the professionals based on their characteristic, i.e., experience in software testing. However, these software testers belong to a single context.

Our theory is generated from the data of a single company only, i.e., the testers of the Company-ICT, which limits the applicability of the theory to dissimilar contexts. However, a properly performed grounded theory approach produces a theory that is flexible and *modifiable* (GT evaluation criterion) [32], [40]. It can be modified using CCM (a key component of GT) based on the data from other studied contexts [31], [32], [40]. With reference to the concept of *biasplexes* (Section 2), confirmation bias belongs to the *inertia* biasplex [17]. The other cognitive biases of this biasplex are, e.g., the bandwagon effect and anchoring bias [17], [38]. These other biases may also reinforce or overlap with confirmation bias among software testers to form their (dis)confirmatory behaviour. Our study is limited to exploring the phenomenon of confirmation bias without considering its biasplex. Furthermore, our study does not employ data triangulation to improve the strength of evidence, instead is limited to a single data collection method, i.e., interviews. However, this study can be considered a post-hoc approach to explore further the phenomenon of confirmation bias among testers because we observed its manifestation in our previous experimental study with student-participants as testers [3].

6 Conclusion and Future Work

We applied grounded theory to explore the antecedents to confirmatory and disconfirmatory behaviours and to understand how they occur among software testers. We identified twenty antecedents to (dis)confirmatory behaviour, classified in nine categories; experience, priority, requirements, functionality retesting, test suite reviews, test execution feedback, time, testing mode and perspective change.

The antecedents that promote confirmatory behaviour, leading to confirmation bias are; ambiguous requirements, incomplete requirements, high priority testing, medium or low priority testing, automated testing, automated test suite reviews, external party reviews, (minor) change/fix size and time pressure. Time pressure plays an important role in the occurrence of confirmatory behaviour among testers, e.g., when they are dealing with ambiguous requirements or performing only a high priority testing.

The antecedents that promote disconfirmatory behaviour and also improve the completeness of a test suite from design and execution perspective are; project experience, past experience, developer and tester perspective, complement testing, detection of errors, absence of errors, no time pressure and internal party reviews. These antecedents may help circumvent confirmation bias and improve the quality of testing. Practitioners are recommended to implement internal party reviews because it may increase the completeness of inconsistent test cases. Similarly, a practice of complement testing, among the testers, may also help in the completeness of inconsistent test cases and increase defect detection. Defect detection (detection of errors), in turn, promotes disconfirmatory behaviour, as propositioned by our grounded theory.

The future work of this study includes the extension and modification of our theory with the data from testers of other companies. In other words, to increase the generality of the theory by sampling other substantive areas. Data triangulation through multiple data collection methods would also increase the validity of findings. For example, conducting an observational study that observes testers over a longer span in addition to interviews would help validate the findings from different sources. This would enable an in-depth analysis of confirmation bias phenomenon, also considering its interactions with other cognitive biases, and thus the behaviour of software testers. Another possible extension of this work is to quantitatively investigate the

relative importance of the identified antecedents, in a software testing context.

ACKNOWLEDGMENTS

The authors would like to thank the participants of the Company-ICT for this study; Alper Corlan, Basak Kahraman, Berkay Sertoglu, Cagatay Ince, Elif Deniz, Emin Vilgenoglu, Gulden Karakoyun, Ozgul Ozcan, Selda Aydin, Sezen Kaya, Sinan Verdi and Ugur Ozcan. This study was supported in part by the Infotech Oulu Doctoral grant at the University of Oulu to Iflaah Salman.

REFERENCES

[1] D. Arnott, "Cognitive biases and decision support systems development: A design science approach," *Information Systems Journal*, vol. 16, no. 1, pp. 55–78, 2006.

[2] L. M. Leventhal, B. Teasley, D. S. Rohlman, and K. Instone, "Positive test bias in software testing among professionals: A review," in *International Conference on Human-Computer Interaction*, 1993, pp. 210–218.

[3] I. Salman, B. Turhan, and S. Vegas, "A controlled experiment on time pressure and confirmation bias in functional software testing," *Empirical Software Engineering*, vol. 24, no. 4, pp. 1727–1761, 12 2018. [Online]. Available: http://link.springer.com/10.1007/s10664-018-9668-8

[4] G. Calikli and A. B. Bener, "Influence of confirmation biases of developers on software quality: an empirical study," *Software Quality Journal*, vol. 21, no. 2, pp. 377–416, 2013. [Online]. Available: http://link.springer.com/10.1007/s11219-012-9180-0

[5] G. Calikli and A. Bener, "Empirical analysis of factors affecting confirmation bias levels of software engineers," *Software Quality Journal*, 2014. [Online]. Available: http://link.springer.com/10.1007/s11219-014-9250-6

[6] L. M. Leventhal, B. E. Teasley, and D. S. Rohlman, "Analyses of factors related to positive test bias in Software Testing," *International Journal of Human-Computer Studies*, vol. 41, pp. 717–749, 1994.

[7] B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why Software Testing Is Sometimes Ineffective: Two Applied Studies of Positive Test Strategy," *Journal of Applied Psychology*, vol. 79, no. I, p. 142, 1994.

[8] G. Calikli and A. Bener, "Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance," in *Proceedings of the 6th International Conference on Predictive Models in Software Engineering - PROMISE '10*, 2010. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1868328.1868344

[9] A. Causevic, R. Shukla, S. Punnekkat, and D. Sundmark, "Effects of Negative Testing on TDD: An Industrial Experiment," in *International Conference on Agile Software Development*, 2013, pp. 91–105. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-38314-4_7

[10] R. Mohanani, I. Salman, B. Turhan, P. Rodriguez, and P. Ralph, "Cognitive Biases in Software Engineering: A Systematic Mapping Study," *IEEE Transactions on Software Engineering*, pp. 1–24, 2018. [Online]. Available: http://arxiv.org/abs/1707.03869https://ieeexplore.ieee.org/document/8506423/

[11] T. Gilovich, D. Griffin, and D. Kahneman, *Heuristics and Biases*, 8th ed. Cambridge University Press, 2002.

[12] A. Tversky and D. Kahneman, "Judgement under uncertainty: heuristics and biases," Oregon Research Institute Research Bulletin, Tech. Rep., 1973.

[13] D. Arnott, "A taxonomy of decision biases," School of Information Management & Systems, Monash University, Australia., Tech. Rep., 1998. [Online]. Available: http://www.sims.monash.edu.au/staff/darnott/biastax.pdf

[14] D. Arnott and S. Gao, "Behavioral economics for decision support systems researchers," *Decision Support Systems*, vol. 122, no. February, p. 113063, 2019. [Online]. Available: https://doi.org/10.1016/j.dss.2019.05.003

[15] D. Kahneman, D. Lovallo, and O. Sibony, "Before You Make That Big Decision...," *Harvard Business Review*, no. June, pp. 51–60, 2011. [Online]. Available: http://website.aub.edu.lb/units/ehmu/Documents/before-you-make-that-big-decision.pdf

[16] I. Salman, "The Effects of Confirmation Bias and Time Pressure in Software Testing," Ph.D. dissertation, University of Oulu, 2019.

[17] P. Ralph, "Possible core theories for software engineering," in *Software Engineering (GTSE), 2013 2nd SEMAT Workshop on a General Theory of*. IEEE, 2013, pp. 35–38.

[18] K. A. de Graaf, P. Liang, A. Tang, and H. van Vliet, "The impact of prior knowledge on searching in software documentation," in *Proceedings of the 2014 ACM symposium on Document engineering*, 2014, pp. 189–198.

[19] E. D. Smith, Y. J. Son, M. Piattelli-Palmarini, and A. Terry Bahill, "Ameliorating mental mistakes in tradeoff studies," *Systems Engineering*, vol. 10, no. 3, pp. 222–240, 2007. [Online]. Available: http://doi.wiley.com/10.1002/sys.20072

[20] G. Calikli, A. Bener, T. Aytac, and O. Bozcan, "Towards a metric suite proposal to quantify confirmation biases of developers," in *International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 363–372.

[21] R. S. Nickerson, "Confirmation bias: A ubiquitous phenomenon in many guises." *Review of General Psychology*, vol. 2, no. 2, pp. 175–220, 1998.

[22] G. Calikli, B. Arslan, and A. Bener, "Confirmation Bias in Software Development and Testing : An Analysis of the Effects of Company Size , Experience and Reasoning Skills," in *22nd Annual Psychology of Programming Interest Group Workshop*, 2010.

[23] G. Calikli, A. Bener, and B. Arslan, "An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers," in *2010 ACM/IEEE 32nd International Conference on Software Engineering*, vol. 2, 2010, pp. 187–190.

[24] S. Eldh, "On Test Design," Ph.D. dissertation, Mälardalen University, 2011.

[25] A. A. Jorgensen and J. A. Whittaker, "How to Break Software," Tech. Rep., 2000. [Online]. Available: https://www.researchgate.net/publication/315700027_How_to_Break_Software_with_examples

[26] J. A. O. G. Da Cunha and H. P. De Moura, "Towards a substantive theory of project decisions in software development project-based organizations: A cross-case analysis of IT organizations from Brazil and Portugal," in *10th Iberian Conference on Information Systems and Technologies, CISTI*, 2015.

[27] J. A. O. Cunha, H. P. Moura, and F. J. Vasconcellos, "Decision-Making in Software Project Management: A Qualitative Case Study of a Private Organization," in *9th International Workshop on Cooperative and Human Aspects of Software Engineering*, 2016, pp. 26–32.

[28] S. Chakraborty, S. S. Sarker, and S. S. Sarker, "An Exploration into the Process of Requirements Elicitation : A Grounded Approach," *Journal of the Association for Information Systems*, vol. 11, no. 4, pp. 212–249, 2010.

[29] I. Hadar, "When intuition and logic clash: The case of the object-oriented paradigm," *Science of Computer Programming*, vol. 78, no. 9, pp. 1407–1426, 2013.

[30] P. Conroy and P. Kruchten, "Performance norms: An approach to rework reduction in software development," in *2012 25th IEEE Canadian Conference on Electrical and Computer Engineering: Vision for a Greener Future, CCECE 2012*, 2012.

[31] C. Urquhart, *Grounded Theory For Qualitative Research: A Parctical Guide*. SAGE Publications, 2012.

[32] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research," in *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. ACM, 2016, pp. 120–131.

[33] S. Baltes and P. Ralph, "Sampling in Software Engineering Research: A Critical Review and Guidelines," *ACM Transactions on Software Engineering and Methodology*, 2020.

[34] H. R. Boeije, "A Purposeful Approach to the Constant Comparative Method in the Analysis of Qualitative Interviews," *Quality and Quantity*, vol. 36, pp. 391–409, 2002.

[35] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding In-depth Semistructured Interviews: Problems of Unitization and Intercoder Reliability and Agreement," *Sociological Methods and Research*, vol. 42, no. 3, pp. 294–320, 2013.

[36] F. Paetsch, A. Eberlein, and F. Maurer, "Requirements Engineering and Agile Software Development," in *Proceedings of the 12th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)*, 2003, pp. 1–6.