What Is a Test Case? Revisiting the Software Test Case Concept

Dani Almog and Tsipi Heart

The Department of Industrial Engineering and Management
Ben Gurion University of the Negev
almog.dani@gmail.com, heart@bgu.ac.il

Abstract. Since the 1980s the term "Test Case" (TC) has been recognized as a building block for describing testing items, widely used as a work unit, metric and documentation entity. In light of the centrality of the TC concept in testing processes, the questions this paper attempts to answer are: What are the uses of TC in software testing? Is there a general, commonly agreed-upon definition of a TC? If not, what are the implications of this situation?

This article reviews and explores the history, use and definitions of TCs, showing that while extensively used in research and practice, there is no one formal agreed upon definition of a TC. In this paper we point at undesirable implications of this situation, suggest four criteria for a 'good' TC definition, and discuss the benefits accrued from such a definition. We conclude by urging the academic and professional community to formalize a TC definition for the benefits of the industry and its customers, and strongly believe that this review paves the way to articulating a formal TC definition. Such a definition, when widely accepted, will clarify some of the ambiguity currently associated with TC interpretation, hence with software testing assessment which relies on TCs as metrics. Furthermore, a formal definition can advance automation of TC generation and management.

1 Introduction

A research initiated by the US Department of Commerce [1] estimated an annual economic damage equivalent to $20 – $52 billion as a result of inadequate software testing infrastructure and processes. The authors classified two primary categories of damages: damages users incurred because of software malfunction, and damages associated with software modification, fixing and re-testing. Although published some six years ago, there is a sound indication that the situation has not significantly improved. Hence, the alarming magnitude of damages caused by inappropriate software testing merits closer investigation into plausible reasons and explanations to this undesirable situation in a quest for solutions and improvement.

Because software testing is a broad topic which cannot be grasped in a single work, this study focuses on one specific aspect of the software testing domain – the test case (TC), since TC is a cornerstone in software testing processes, and because, as shown later on, it is posited that inconsistencies in TC definitions and use throughout the testing process is perhaps a cause for fundamental flaws.

The questions this paper attempts to answer are: What is the role of TC in software testing? Is there a general agreement about the definition of TC? If not, what are the consequences of this situation?

R.V. O'Connor et al. (Eds.): EuroSPI 2009, CCIS 42, pp. 13–31, 2009.
© Springer-Verlag Berlin Heidelberg 2009

We believe that answering these questions will clarify some of the ambiguity currently associated with TC interpretation, and pave the way to articulating a formal TC definition. If and when widely accepted, it can relieve some of the ambiguity associated with software testing metrics that commonly relies on counting TCs. Furthermore, an appropriate formal definition can drive automation of TC generation and management. Therefore, this work is clearly a contribution to software process improvement by dealing with an important aspect of testing – the test case.

The rest of the paper is organized as follows: common software testing processes and practices are briefly described in the next chapter, showing the importance of testing processes in software engineering, and the TC as the testing building block. We then describe the literature survey methodology employed. Next, several definitions for TCs are presented as a result of the literature survey, showing the conceptual variability of these definitions. We then proceed to a review of the literature discussing the centrality of TCs in testing processes, concluding with a suggestion of dimensions by which a TC definition can be evaluated, as well as an evaluation of existing definitions based on these dimensions. The paper concludes with a discussion of the implications of the lack of a unified approach to TCs and whether there is a need to re-define this term.

2 Common Practices in Software Testing

In the following section the importance of testing in terms of its substantive role in software development on the one hand, and of its complexity, on the other hand, is briefly presented. This background clarifies the merit in further looking into TC use and definitions, since TCs are building blocks of testing.

The testing effort undoubtedly comprises a significant portion of the programming effort. For example, an early research conducted at NASA [2] found that testing efforts comprise 30% of the time invested by programmers, and 37% of their actual work days (Figure 1).

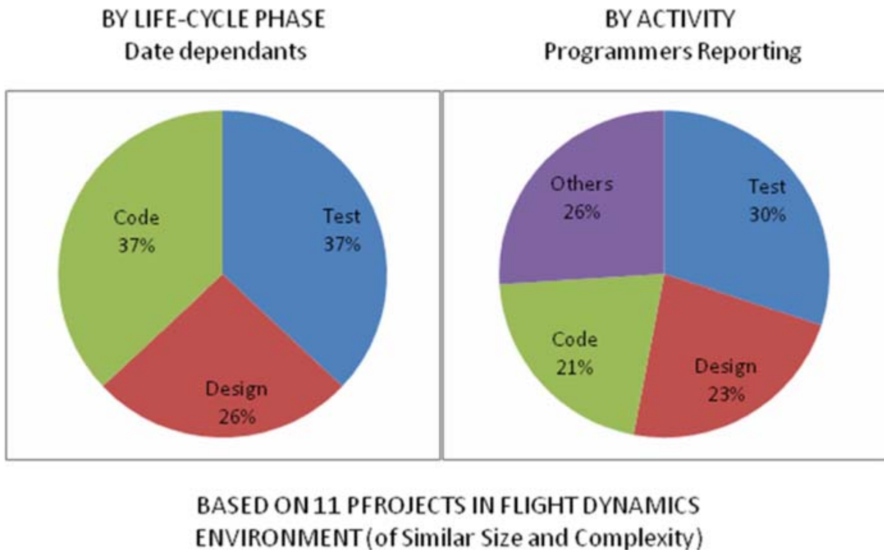

Fig. 1. Distribution of the effort among programmers' tasks (NASA) [2]

A more recent study conducted in Alberta, Canada [3] identified large variance among projects regarding testing resources in terms of the ratio of developers to testers, showing that about 50% of the studied projects allocated around two developers to one tester (~50%), whereas 35% invested much less personnel resources in testing (five developers to one tester, ~20%). Other studies generally support these findings, substantiating the positive correlation between software development process maturity and the degree of investment in software testing – around 35% of the overall investment [4, 5].

Testing tasks have been traditionally classified into three phases [6]: 1) Preparation: plan, design, construct, 2) Execution, and 3) Verification: verify results against expected outcomes and report. These three stages were often performed sequentially as in structured software development process models, demanding rather equal resource investment. Recently, however, there is a tendency to change this structured model due to several reasons [7]. One reason is the growing popularity of new software development models and techniques, such as agile methods, service oriented architecture (SOA), and test driven development (TDD), all three indicating testing processes that somewhat deviate from the structured process models. Along changes in development models, testing automation has matured and is now more prevalent, potentially easing the execution phase. Finally, verification and validation processes become more complex due to the growing complexity of the developed applications and the data units involved. For example, growing complexity can be attributed to data representation simultaneously using various techniques as databases, XML files, encryption, compression, coding, dynamic data location, etc. Consequently, a deeper understanding of the data structure and characteristics is required during testing, as well as more sophisticated tools and processes.

In light of the growing complexity of the testing process, Bach [8] advocated exploratory testing, defined as “any testing to the extent that the tester actively controls the design of the tests as those tests are performed and uses information gained while testing to design new and better tests” (p. 2). This methodology addresses the assertion that complete testing preparation is unlikely at an initial phase of the testing process. Thus, Kaner [9] explained the advantages of exploratory testing in allowing testers to learn while they test, to get more sophisticated as they learn, interpret and design their tests differently as they learn more about the product, the market, the variety of uses of the product, the risks, and the mistakes that are likely to be made by the humans who wrote the code. Under exploratory testing the test plan evolves during the test development and execution, rather than pre-planned before the actual complexity of the product is realized. This realm, however, might be practically problematic when having to pre-estimate testing efforts as part of the overall project estimation. Evidently, there is a broad agreement that testing is a complex task, hence difficult to estimate and quantify. In-depth examination of various testing processes and techniques is beyond the scope of this work, instead, we focus on the common building block of all software testing techniques – the TC. Thus, in order to better understand the problem at hand, we next bring a review of the literature, elaborating on the single concept common to all testing processes and techniques – the TC.

3 Methodology

The following methodology has been employed in order to identify the literature relevant for this review. First, Google and Google Scholar were used as search engines to find sources with the keywords: "software testing", "software reliability", "testing methodology", "black box testing", and "test case". This first search effort yielded about 150 papers and about 25 books dated 1982 onwards, which were scanned for relevance by reading their abstracts. Looking at citations appearing in the elicited papers implied that there is merit in further expanding the search by using the following keywords: "TDD", "SOA", "agile", "software cost estimation", "software project management", "testing projects", "test case generation", and "testing automation". This search yielded about 100 additional papers and about 10 books spanning the years 1980 till 2008.

A similar search has been conducted on leading journals and conferences, for example relevant IEEE and EMSE journals and ICST conferences that directly or indirectly included topics represented by the above keywords. These three rounds of literature search resulted in a database of about 300 papers, books, and conference proceedings. Endnote 9.0 has been used as the reference management tool, where research notes have been added for classification purposes.

This reference database has been then reviewed, and each reference has been classified to sub-topics as in Table 4 (a paper could be related to more than one subtopic), as well as whether or not it included a formal definition of a TC. Those papers which contained such a definition were further categorized based on the nature of the definition, as appears in Table 3.

While classifying the papers, additional references and topics were searched by scanning their reference list, which resulted in about 40 additional papers, bringing the total number of papers and books reviewed to about 340, of which 267 directly referred to TCs.

4 Literature Review

4.1 Historical Overview of the TC Concept

The TC concept appeared as a central concept underlying testing processes since the beginning of formal software testing, for example as part of the Systematic Test & Evaluation Processes (STEP) model [10], which defined feedback loops between software development and testing. Three sources for TC generation were identified: directly from the requirements, stemming from performance requirements, and based on system's design [10]. A formal definition of a TC, however, was not included. In a study published in the same year, Ostrand & Balcer [11] suggested to build TCs as a collection of test frames and test scripts, yet these two terms were not precisely defined although TCs were perceived to be measureable by their size. Weyuker [12] brought a quite different approach when she maintained that TCs are formed by decision statements, and recognized that the more the number of decision statements in the tested code, the more complex is the TC, recommending to limit the average

number of decision statements tested by one TC to 3.6. Interestingly, in spite of frequent use of the term TC in her paper (76 times) it was not formally defined.

The centrality of the TC is evident in the work of Harrold, Gupta & Soffa [13], who used TCs as the basis of a methodology to minimize testing efforts, realizing that the testing process could in fact become indefinite because of the lack of indicators for absence of errors. They developed a structured methodology to identify redundant TCs and merge them into TC suites or execute these TCs in pairs. In this work TCs were identified as *TC requirements* assuming that TCs stem from requirements. Adopting an analogous line of thinking, Rosenberg, Hammer & Huffman [14] maintained that TC content should reflect the requirements, and therefore should be controlled by a TC coverage matrix, which maps requirements to TCs, aimed at optimizing the testing effort. Clearly, TCs and the resulting coverage matrix tend to become more complex relative to the number and complexity of the requirements. In an effort to handle this growing complexity, Iberle [15] developed a TC hierarchy methodology at HP labs, where the test plan was formed by test groups based on the system's functionality as defined by the requirements, the system's design and other sources, and each test group is then further detailed into tests composed of TCs in the leaves (Figure 2). Here again, the TC was the fundamental building block of the testing process, yet no formal definition was provided.

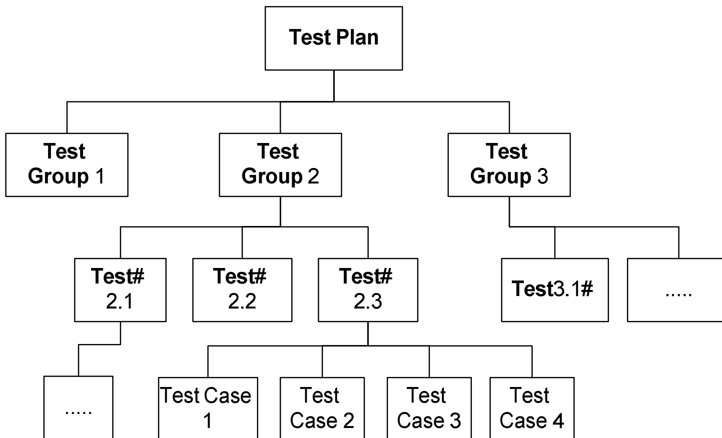

Fig. 2. Test plan hierarchy [15]

Aichernig [16] was among the few researchers who attempted a formal TC definition by developing a mathematical description of a TC, although he suggested that TCs were in fact abstractions of the requirements, or "highly abstract contracts" (p. 6). Aichernig's mathematical approach to TC definition (brought later on in section 4.4) was aimed at advancing a formal language essential for automation of TC generation.

In the first chapter of his book "Software Testing: A Craftsman Approach" Jorgensen [17] reviewed the TC concept, noting that the TC was the key to the success of the testing process. He distinguished between TCs identified by the functional requirements (functional testing) and TCs identified by the software structure (structural testing).

In an attempt at identifying "what is a good test case?" Kaner [18] maintained that a good TC was one that gave the required *information* which was the objective of the particular test. He counted several testing objectives each requiring a different type of TCs, and acknowledged that TCs greatly vary and hence using them as metrics is problematic: "Also, under this definition, the metrics that report the number of test cases are meaningless. What do you do with a set of 20 single-variable tests that were interesting a few weeks ago but now should be retired or merged into a combination? Suppose you create a combination test that includes the 20 tests. Should the metric report this one test, twenty tests, or twenty one?" (p. 2).

Later works by Grindal and Colleagues [19, 20] included a review of mechanisms to render software testing more efficient and effective, heavily relying on TC selection and execution, since they maintained that testing is "loosely considered to be the dynamic execution of test cases" [19, p. 2]. An interesting approach has been adopted by the aerospace industry where the Conformance and Fault Injection (CoFI) methodology has been used [21, 22]. Under this methodology, TCs were differentiated between those that aim at confirming the appropriate behavior of the tested product and those that are aimed at creating faulty situations. The authors suggested a structured approach to the definition of the two types of testing, and as a result, a systematic creation of the relevant TCs.

Because of the centrality of the TC in the testing process, and due to the significant effort invested in designing and generating TCs especially in large or complex projects, several studies have elaborated on TC management processes and tools. For example, Desai [23] from Bell Laboratories described a tool which managed the configuration and inventory of TCs separately from the testing tasks, compatible with the IEEE 829 standard. A later work described a TC management and tracking tool, where the term 'test item' is used in a context similar to TC [24], making the TC concept even more ambiguous in the absence of a formal definition. The need to automate the generation and management of TCs was demonstrated in Jorgensen's [25] work, where he noted that it took 141,306 TCs to test version 5.0.1 of Acrobat Reader. It is noteworthy that Jorgensen did not define a TC unit in this work as the basis for the counting method although the term was extensively used in this commentary.

The likely variability among TCs has been acknowledged by Nagappan [26] who developed the Software Testing and Reliability Early Warning (STREW) metric suite for software testing effort estimation, using TCs as one of the model metrics. He warned, however, that using TCs as a metric might not be well defined since "….one developer might write fewer test cases each with multiple asserts checking various conditions. Another developer might test the same conditions by writing many more test cases, each with only one assert" (p. 39). This variability among TCs should be taken into account when defining effort estimation model parameters. Table 1 shows that TCs can greatly vary, for example by complexity, size (whether containing many asserts or one assert), or by origin (requirements or other), hence cannot be unified as indicating a singular metric.

Table 1. Software rating – defect density, [27]

Rating	Very Low Defect Density	Very High Defect Density
Test Cases	Few test cases	Many test cases
Test case asserts	Asserts that only exercise "success" behavior of the product or do not adequately cover the functionality of the product	Asserts that exercise various behaviors of each requirements
Requirements	Test cases do not relate to requirements	At least one test case per requirement
Code coverage	Minimal coverage of important functions	100% coverage

A further warning in this regard has been advocated by Hoffman [28], who pointed at the possibility that definitions of TCs, as well as their number and content, might change during the course of the project, jeopardizing the validity of metrics based on these TCs.

4.2 TC Use and Generation in Modern Software Development

Not only have TCs been important in traditional software development processes, they also continue to play an important role in more modern software development methodologies and techniques.

Similar to the more traditional software development environments, the TC is a fundamental entity in testing software in the *object oriented* environment. For example, Binder [29] first developed a methodology for TC generation in an object oriented environment, by introducing the 'testing points' concept, a mechanism used to define test requirements and the relevant TCs. Later in his book Binder suggested to define the TC as a method thereby including the test itself as part of the design of the objects.

Agile software development methods have quite revamped traditional testing concepts, particularly the division between testers and developers [30], since on-going testing is one of the principles guiding development of very small and frequent software iterations common to the agile methodology. Nonetheless, the centrality of the TC concept has not changed as a result of utilizing these methodologies, although the test planning method has.

TDD or TDM are software development methods that advocate writing TCs prior to the actual software development to assure developing software that is testable [31, 32]. Here, the role of the TC is even magnified, yet evidence about the effectiveness of this method is still mixed [31, 33].

Service oriented architecture (SOA) has introduced new testing challenges [34] demonstrated for example by the inclusion of a testing mechanism in the SOA infrastructure delivered by IBM [35]. Especially challenging is testing composed and complex services that require new testing methods [36], making estimation of testing scope and effort more difficult. The recent move to SOA has raised the interest in

software componentization [37, 38] and component-based testing, adding additional ambiguity to the TC concept.

Some research has focused on *automatic TC generation*, a process requiring TC formalization [39-42]. As use cases largely reflect functional requirements in the UML environment, Nebut, Fleurey, Le Traon & J'z'quel [43] suggested TC generation from use cases, after incorporating the contract element they claim is a component essential for translating a use case into a TC. Likewise, test objectives and sequence diagrams also serve as sources for TC generation. Generally, several works have developed techniques to generate TCs from UML diagrams, termed Model Based Testing (MBT), mostly based on transforming use cases and states into TCs [44, 45]. Although the attempts to automate TC generation resulted in some level of formalization, the difficulties pertaining to the TC concept were not solved by this mechanism, since use cases and scripts all suffer from the same fuzziness of definition regarding size, complexity, number of states, etc.

4.3 TCs as Metrics

During the testing phase, there is a need to manage and control the process, by measuring its size, complexity, and quality, as a minimum. This, however, is easier said than done, due to reasons brought in the previous sections. Thus, for example when using the Goal – Questions – Metrics (GQM) method [1]developed by V. Basili and D. Weis for measurement development, Management strives to find metrics to answer questions such as 'how long would it take to complete testing?', or 'how much resources should be allocated to testing?', aimed at achieving managerial goals such as appropriate resource allocation and adhering to schedules. Measures developed to answer these questions often rely on number of TCs, for example "total number of planned white/black box test cases run to completion, number of planned integration tests run to completion, or number of unplanned test cases required during the test phase" [26, p. 15]. The Software Testing Reliability Early Warning Model for Java (STREW-J) developed by Nagappan [26] to estimate expected problems as a means to estimate testing efforts used at least two estimation parameters that are based on number of TCs: 1) *number of test cases divided by source lines of code (R1)* as an indication of whether there are too few test cases written to test the body of source code; and 2) *number of test case divided by number of requirements (R2)* as an indication of the thoroughness of testing relative to the requirements. Other TC-based metrics recommended as reflecting the status of the testing project were number or percent of TCs run since testing started, number or percent of TCs run since the last status report, number of percent of TCs that passed since the beginning of the testing project, number or percent of TCs passed since the last status report, number or percent of failed TCs, total number of open issues or TCs not run [46].

Elsewhere, eight of thirteen reports recommended as tools for testing monitoring and control were based on TCs count, completion status, results etc. [47]. Further, these same authors suggested eighteen indicators to monitor the project status, eleven of which are based on tests or TCs. Two real-world examples of using TCs as the unit for testing progress monitoring are presented in Figures 4 and 5. Figure 4 illustrates

[1] We thank the reviewer for suggesting using GQM as a metric-generation methodology.

NASA's recommendation for test execution monitoring, and Figure 5 was drawn from a real-world project at a large telecom enterprise, where three different projects were tracked based on the number of TCs not yet executed (test backlog). Evidently, not only all TCs were equally counted, but also TCs from different projects were compared under the same unit of analysis, regardless of potential variance among TCs stemming from the dissimilarity of the projects.

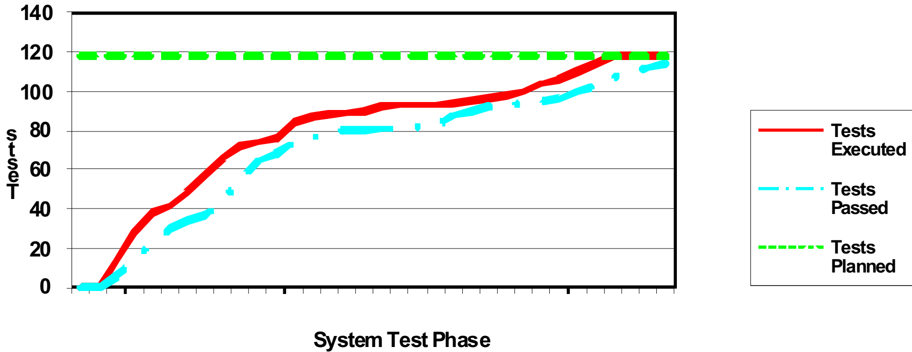

Fig. 3. Testing execution progress monitoring, [48]

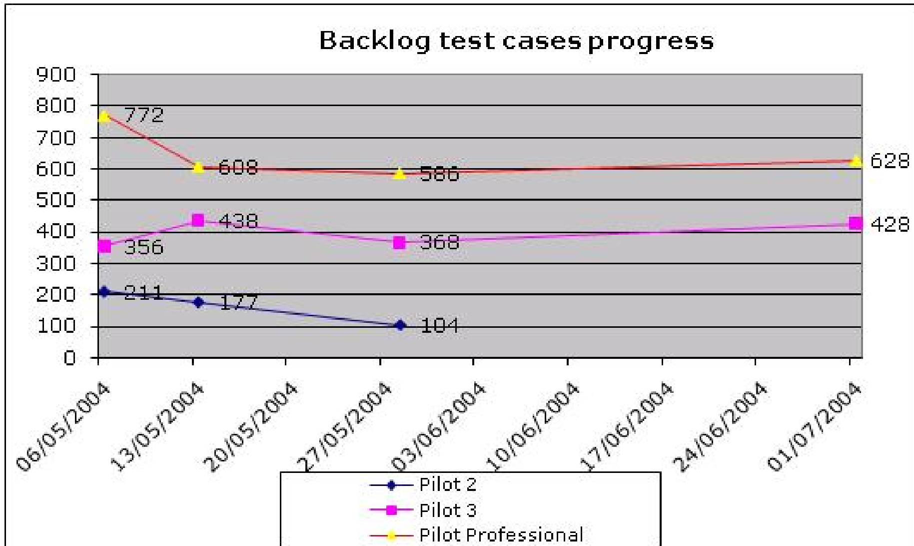

Fig. 4. Testing execution progress monitoring

In the next example (Table 2), number of tests was recommended as a metric to track and control testing execution. Since tests are composed of TCs it is reasonable to assume that this metric implies actually counting TCs from different software features ignoring their likely differences.

Table 2. Number of tests is used as a metric for testing monitoring and control [46]

Project online trade				Date: 5/23/2007	
Feature tested	Total Tested	# Complete	% Complete	# Success	% Success
Open Acct	46	46	100	41	89
Sell Order	36	25	69	25	69
Buy Order	19	17	89	12	63
……					
Totals	395	320	81	311	79

Similarly, IBM published reporting metrics for testing the software developed by various vendors under IBM's supervision for the Sydney Olympic Games, all based on counting number of TCs [49]: 1) Number of test cases defined, 2) Number of test cases executed, 3) Number of test cases with failures but no associated defect records 4) The percentage of test cases attempted, used as an indicator of progress relative to the completeness of the planned test effort.

TCs has also been used for testing effort estimation in few works where overall project effort has been estimated based on distinctive estimation of the various development phases [50-52]. In an attempt to overcome the problem of counting TCs of various size and complexity Nageswaran [53] suggested using function points where the number of TCs can be determined by the function points estimate for the corresponding effort. Following this approach Aranha & Borba [54] presented a scheme for collecting execution points for calculating and estimating testing efforts. It should be noted, however, that none of these works formally defined the TC term although.

Evidently, TCs have been used as metrics for testing effort estimation, as well as for testing monitoring and control. Common to most of the techniques suggested in these works is the reliance on counting TCs, with only minimal reference to the fact that TCs lack a standard definition and tend to greatly differ.

4.4 Test Case Definitions

As stated earlier, a thorough literature survey has been conducted in order to study where and how TCs are defined. Interestingly, in spite of a plethora of research about software quality assurance, few works formally define a TC, although most use this term quite intensively. Perhaps most notable is the fact that an explicit definition of a TC could not be located in the 2004 version of SWEBOK. Rather, the TC appears as an integral part of the general software testing definition:"ïSoftware testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite executions domain, against the expected behavior" [55, p. 5-1]. Nonetheless, several definitions have been retrieved, classified into four dominant approaches: 1) input-process-output-objectives, 2) states and transitions, 3) contractual approach, and 4) other definitions.

The *input-process-output-objectives* perspective conceptualizes a TC as a set of inputs into a pre-defined process, aimed at yielding a desired output, based on the test

Table 3. Test Case Definitons and Sources

Category	Definition	Source
Input-Process-Output-Objectives	"A set of conditions or variables under which a tester will determine if an application or a software system meets specifications…. It may take many test cases to determine that a software program or system is functioning correctly"	www.wikipedia.org
	"A test case is the combination of test data and oracle information to determine the validity of the test"	[56, p. 9]
	"A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement"	[24, p. 187]
	"Test case is a test vector consisting of a set of test inputs and the corresponding test outputs (pre and post conditional assertions)"	[45, p. 2]
	"Test Case is an identified set of information including inputs and expected outputs associated with a particular program behavior"	[17, p. 7]
	"A test case is a finite structure of input and expected output: a pair of input and output in the case of deterministic transformative systems, a sequence of input and output in the case of deterministic reactive systems, and a tree or a graph in the case of non-deterministic reactive systems	[32, p. 2]
States and Transitions	"A sequence of one or more subtests executed as a sequence because the outcome and/or final state of one subtest is the input and/or initial state of the next. The word 'test' is used to include subtests, tests properties, and test suites".	[57, p. 13]
	"A test case specifies the pretest state of the implementation under test (IUT) and its environment, the test inputs or conditions, and the expected result. The expected result specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, returned values, and resultant state of the IUT and its environment. Test cases may also specify initial and resulting conditions for other objects that constitute the IUT and its environment."	[29, p. 47]
	"Test case is composed of several components: test case values, prefix values, verify values, exit commands and expected outputs"	[58, p. 28]
	"Test Case is a verification of some aspect of the System Under Test (SUT). Test Case for any feature of any SUT can be defined as follows: Perform verification, Vv Which may be preceded by a sequence of actions, Aa Which may require a set of data, Dd	[59, p. 51]

Table 3. (*continued*)

	Which may require preconditions, Pp All of which runs in environment, Ee Hence, a Test Case, Tt = Ee Pp Dd Aa Vv"	
Contract	"Test-cases common in software engineering are in fact contracts (highly abstract contracts)… However, our result that test-cases are abstractions holds for general contract statements involving user inter-action".	[16, p. 8]
	"a form of contract between a service provider and a service user"	[60, p. 2]
Other	"An empirical frame of reference, rather than a theoretical one"	[61, p.359]
	"…test case is a question that you ask of the program. The point of running the test is to gain information, for example, whether the program will pass or fail the test"	[18, p. 2]
	"A test idea is a brief statement of something that should be tested. For example, if you're testing a square root function, one idea for a test would be 'test a number less than zero'. The idea is to check if the code handles an error case"	[18, p. 2]
	"a specific set of attribute values that tests a given logical situation"	[62, p. 3]
	"a test case can be considered as a predator while a mutant program is analogous to a prey"	[63]

objective. The *states and transitions* approach considers a TC as a set of transition patterns among states. The *contractual* approach defines TC as a contract since the outcomes of pre-defined conditions are fully defined. Finally, there are several *other definitions* stemming from various contexts. Table 3 lists examples of definitions in each category. The implications of this variability are discussed next.

5 Discussion

The TC serves as the backbone of testing processes, and is a fundamental unit for test planning, execution, monitoring and control. It is also used as a common metric in quantifying testing effort, scope and status. Furthermore, there is a growing quest to automate TC generation, execution and management. Nonetheless and quite interestingly, there is no consensus regarding the formal definition of a TC.

From the papers reviewed for this work it is evident that the TC concept is frequently used in various contexts, yet infrequently formally defined (Table 4). Please note that numbers in Tables 4 and 5 do not add up because papers could be classified to more than one sub-topic.

Table 4. TC-related papers, definitions and contexts used

Topic	Cost/ROI Estimations	Management	UML/MBT OO/SOA	Metrics	Automation/ Generation	Total
TCs Papers Reviewed	86	69	46	25	44	267
Formal TC Definition	11 (13%)	19 (28%)	26 (57%)	4 (16%)	14 (32%)	38 (14%)

Table 4 shows that 267 reviewed papers referring to TCs covered five different topics, yet only in 38 papers (14%) a formal definition of TC was attempted, particularly in studies focusing on OO related issues and TC automation and Management. It is thus valid to wonder why only 14% of authors bothered to formally define the central concept of their work in spite of heavily using this term (some mention TC more than a hundred). Thus, in the 38 papers where TC was defined, various definitions were employed representing all four definition categories: input-process-output-objective, states & transitions, contract, and other. It is thus interesting to examine whether there is an association between the definition category used and the specific context (Table 5). For example, it could be expected that works in the UML/MBT/OO context would use states & transitions definitions that stem from the OO world. This, however, could not be substantiated by the present literature review, as those few authors who have used the TC definition in their OO-related work chose definitions from all categories (Table 5). Moreover, no author has articulated the reasons for choosing one definition or another. As seen in Table 5, authors using TCs in the context of OO/MBT/UML more frequently used the input-process-output-objective (termed hereinafter process-based for brevity sake) definitions rather than the more naturally related states & transitions definitions, which turn out as the most popular definition category. Evidently, no correlation could be deduced between the definition category and the context, possibly attesting to the arbitrary choice of the former.

Table 5. TC definition distribution by research context

Context	Definition category				
	Process	States	Contract	Other	Defined
Cost/ROI	6	1	1	3	11
Management / Project	11	4	1	3	19
OO/ MBT/UML/SOA	12	6	2	6	26
Measurements/Metrics	1	1	1	1	4
Automation/Regression	10	1	1	2	14
Total	18	11	2	8	38

The lack of formal TC definition and the fact that most studies do not include any definition raise several questions: Is such a definition required? Are there deficiencies in the existing definitions? What are the implications of the lack of a formal definition?

We maintain that a formal definition is indeed required, encouraged by the fact that in real-world testing of life-threatening projects a formal definition is an important part of the testing guidelines. For example, based on the IEEE standard, chapter 6 of a manual for testing safety applications in a nuclear reactor environment greatly elaborates on TC types, definitions, content, and documentation [64]**.** Four types of TCs are specified: 1) verification TC, 2) validation TC, 3) demonstration TC, 4) general suitability TC. Each TC is defined by a general description including reference number, geometry, flow features, experimental data, existing simulations, related experiments, and rating of the challenge the test case poses. These details should be accompanied by further documentation describing the test environment for each TC.

It is suggested that a formal TC definition could render several benefits if satisfying at least four requirements: 1) Unambiguousness: such TCs would be uniformly understood by the various stakeholders participating in a testing endeavor, 2) Generalizability: TCs would hold upon transforming from one platform to another, from one testing domain to another, and so on, 3) Quantifiability: only quantifiable TCs would be sensibly measured, and 4) Automatability: some might argue that this trait is an outcome of the above three characteristics, yet we chose to explicitly indicate it as a desirable feature because of its importance.

Unambiguousness ensures a unified view shared by all professionals involved in software testing regardless of their prior experience, background, testing environments, methods and techniques. This trait is important because it will ease the current 'Tower of Babylon' dominating the testing world, and rive sharing expertise among various testing schools and perceptions. *Generalizability* ensures maintaining testing assets and investments along various testing efforts, namely, TC generation tools and techniques would be valid in different testing environments. *Quantifiability* is clearly beneficial because of the importance of the TC as a fundamental metric. Currently, measurements involving counting TCs are clearly inconsistent. Finally, there is no need to explain the benefits rendered by the ability to *automate* TC generation, execution and management. Several attributes are mandatory for TC automation, among them is a formal definition of the TC structure.

Examining the existing definitions by the four categories illustrates the deficiencies in each type. The *input-process-output-objective* definitions are generally unambiguous,

but not necessarily generalizable. For example, non-functional requirements, such as testing a user experience, are difficult to define using this type of definition. Likewise, the 'process' part of the TC can vary in size and complexity hence difficult to quantify and measure. For instance, a process can be as simple as 'check for existence of a certain value' or quite complex as 'create a customer order'. Consequently, this type of definition is problematic to automate. The *state & transitions* definitions may satisfy the unambiguousness and quantifiability traits but are hardly generalizable since they stem from the state-machine world, therefore not transferrable to other testing domains. For example states and transitions that are a result of dynamic environmental conditions and data would be rather impossible to define as a finite number of states and transitions. TCs defined as States & Transitions, however, are quite convenient to quantify and automate due to their origination in the state-machine domain. The *contract* group of definitions is becoming popular, mainly in SOA platforms, yet these definitions clearly violate the unambiguousness criterion. For example, Aichernig [16] defined a test as a contract between the user and the software provider, Mikhailova et al. [65] defined testing as a contract between the system under test and its environment, and Bruno et al. [66] thought it was a contract ensuring service compliance between releases. Clearly, only a formal definition of the contract, such as the one attempted by Aichernig [16] is unambiguous. For similar reasons it cannot be generalized, quantifiable or automatable unless formalized. Finally, it is quite obvious that the *other* definitions do not meet most of the above requirements.

We maintain that the absence of a formal definition for TCs causes test planning, execution, and monitoring malfunctioning. For example, reporting testing effort estimation or testing progress by number of executed TCs is clearly misleading, often resulting in projects not meeting time and budget constraints, or in inadequate software quality. Testing automation efforts are likewise contingent upon formal definition of TCs, hence its absence is possibly one of the barriers to a broader diffusion of automation tools. These shortcomings are quite likely among the causes for the huge annual economic damage as a result of inadequate software testing infrastructure and processes reported by the US Department of Commerce [1]. Hence, further work towards a formal TC definition that meets the above requirements is advocated.

6 Conclusions

TC is a cornerstone for planning, designing, and monitoring testing projects, as well as a means for work, effort and cost estimation. This work demonstrated not only the centrality of the TC but also the variance among TC definitions. Further, the official professional taxonomies, for example those presented in the joint ISO-IEEE Guide to the Software Engineering Body of Knowledge – SWEBOK does not explicitly define TC.

This situation is possibly a barrier to improving the testing infrastructure leading to higher software quality, therefore decreasing the enormous resulting damage. It is suggested that establishing a formal, unambiguous, generic, quantifiable and structural definition for a TC would be a significant contribution to the world of software testing, and software quality in general. Such a definition would pave the way to standard TC generation techniques, as well as to measurement and evaluation tools.

Referring to Kaner's [18] question "what is a good Test case?" and his assertion that "good TC is one that gives the required information", we see benefits in formalizing a unified, well defined and structured TC entity that satisfies all the above dimensions. We suggest pursuing, determining and proposing an improved and comprehensive definition of a test Case.

References

[1] Tassey, G.: The Economic Impacts of Inadequate Infrastructure for Software Testing. National Institute of Standards and Technology (2002)
[2] Basili, V., Caldiera, G., McGarry, F., Pajerski, R., Page, G., Waligora, S.: The software engineering laboratory: an operational software experience factory. In: Proceedings of the 14th international conference on Software engineering. ACM, New York (1992)
[3] Geras, A.M., Smith, M.R., Miller, J.: A survey of software testing practices in alberta. Canadian Journal of Electrical and Computer Engineering 29(3), 183–191 (2004)
[4] Grindal, M., Offutt, J., Mellin, J.: On the Testing Maturity of Software Producing Organizations. In: Proceedings of the Testing: Academic and Industrial Conference-Practice and Research Techniques TAIC PART (2006)
[5] Ng, S.P., Murnane, T., Reed, K., Grant, D., Chen, T.Y.: A Preliminary Survey on Software Testing Practices in Australia. In: Software Engineering Conference (2004)
[6] Illes, T., Herrmann, A., Paech, B., Rockert, J.: Criteria for Software Testing Tool Evaluation. A Task Oriented View. In: Proceedings of the 3rd World Congress for Software Quality (2005)
[7] Almog, D.: Verification Points for Better Testing Efficiency. In: StarEastSQE (2007)
[8] Bach, J.: Exploratory Testing Explained, `http://www.satisfice.com/articles/et-article.pdf`
[9] Kaner, C.: The Ongoing Revolution in Software Testing. In: Software Test & Performance Conference (2004)
[10] Gelperin, D., Hetzel, B.: The Growth of Software Testing. Communications of the ACM 31(6), 687–695 (1988)
[11] Ostrand, T.J., Balcer, M.J.: The Category-Partition Method for Specifying and Generating Functional Tests. Commun. ACM 31(6), 676–686 (1988)
[12] Weyuker, E.J.: The Cost of Data Flow Testing: An Empirical Study. IEEE Transactions on Software Engineering 16(2), 121–128 (1990)
[13] Harrold, M.J., Rajiv, G., Mary Lou, S.: A Methodology for Controlling the Size of a Test Suite. ACM Trans. Softw. Eng. Methodol. 2(3), 270–285 (1993)
[14] Rosenberg, L., Hammer, T.F., Huffman, L.L.: Requirements, Testing and Metrics. In: 15th Annual Pacific Northwest Software Quality Conference (1998)
[15] Iberle, K.: Divide and Conquer: Making Sense of Test Planning. In: The International Conference on Software Testing, Analysis and Review, STARWEST (1999)
[16] Aichernig, B.K.: Test-Case Calculation through Abstraction. In: International Symposium of Formal Methods. Springer, Heidelberg (2001)
[17] Jorgensen, P.: Software Testing: A Craftsman's Approach. CRC Press, Boca Raton (2002)
[18] Kaner, C.: What Is a Good Test Case? In: Star East (2003)
[19] Grindal, M., Offutt, J., Andler, S.F.: Combination Testing Strategies: a Survey. Software Testing Verification and Reliability 15(3), 167 (2005)

[20] Grindal, M., Lindstrom, B., Offutt, J., Andler, S.F.: An Evaluation of Combination Strategies for Test Case Selection. Empirical Software Engineering 11(4), 583–611 (2006)
[21] Ambrosio, A., Mattiello-Francisco, F., Santiago, V., Silva, W., Martins, E.: Designing Fault Injection Experiments Using State-Based Model to Test a Space Software. In: Bondavalli, A., Brasileiro, F., Rajsbaum, S. (eds.) LADC 2007. LNCS, vol. 4746, pp. 170–178. Springer, Heidelberg (2007)
[22] Ambrosio, A.M., Martins, E., Vijaykumar, N.L., de Carvalho, S.V.: Systematic Generation of Test and Fault Cases for Space Application Validation. In: DASIA: Data Systems in Aerospace, European Space Agency (2005)
[23] Desai, H.D.: Test Case Management System (TCMS). In: IEEE Conference Global Telecommunications GLOBECOM: 'Communications: The Global Bridge' (1994)
[24] Craig, R.D., Jaskiel, S.P.: Systematic Software Testing. Artech House (2002)
[25] Jorgensen: Testing with Hostile Data Streams. ACM Sigsoft Software Engineering Notes 28(2), 1 (2003)
[26] Nagappan, N.: A Software Testing and Reliability Early Warning (STREW) Metric Suite, Thesis: Computer Science, North Carolina University (2005)
[27] Sherriff, M., Boehm, B.W., Williams, L., Nagappan, N.: An Empirical Process for Building and Validating Software Engineering Parametric Models. North Carolina State Univeristy CSC-TR-2005-45, October, 19 (2005)
[28] Hoffman, D.: The Darker Side of Metrics. In: Conference of the Association of Software Testing, CAST (2006)
[29] Binder, R.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Addison-Wesley Professional, Reading (2000)
[30] Talby, D., Hazzan, O., Dubinsky, Y., Keren, A.: Agile Software Testing in a Large-Scale Project. IEEE Software, 30–37 (2006)
[31] Beck, K.: Test-driven Development: By Example. Addison-Wesley Professional, Reading (2003)
[32] Utting, M., Legeard, B., Pretschner, A.: A Taxonomy of Model-based Testing. Dept. of Computer Science, University of Waikato Hamilton, New Zealand (2006)
[33] Bohnet, R., Meszaros, G.: Test-Driven Porting. In: Proceedings of the Agile Development Conference (2005)
[34] Lewis, G.A., Morris, E., Simanta, S., Wrage, L.: Common Misconceptions about Service-Oriented Architecture. In: Proceedings of the Sixth International IEEE Conference on Commercial-off-the-Shelf (COTS)-Based Software Systems (2007)
[35] Hiebert, D., Klaedtke, R.A., Lowery, E., Nartovich, A., Raut, N., Sandberg, M.J.: Building SOA-based Solutions for IBM System i Platform. IBM (2007)
[36] Karam, M., Safa, H., Artail, H.: An Abstract Workflow-Based Framework for Testing Composed Web Services. In: IEEE/ACS International Conference on Computer Systems and Applications, AICCSA (2007)
[37] Rehman, M.J., Jabeen, F., Bertolino, A., Polini, A.: Testing Software Components for Integration: A Survey of Issues and Techniques. Software Testing, Verification & Reliability 17(2), 95–133 (2007)
[38] Weyuker, E.J.: Testing Component-Based Software: A Cautionary Tale. IEEE Software 15(5), 54–59 (1998)
[39] Cai, K.Y., Zhao, L., Hu, H., Jiang, C.H.: On the Test Case Definition for GUI Testing. In: Fifth International Conference on Quality Software, QSIC (2005)
[40] Boujarwah, A.S., Saleh, K.: Compiler Test Case Generation Methods: A Survey and Assessment. Information and Software Technology 39(9), 617–625 (1997)

[41] Calam, J.R., Ioustinova, N., Pol, J.: Towards Automatic Generation of Parameterized Test Cases from Abstractions. Technical Report SEN-E0602, Centrum voor Wiskunde en Informatica (2006)
[42] Byers, D., Engstrom, M., Kamkar, M.: The Design of a Test Case Definition Language. Automated and Algorithmic Debugging, 69–78 (1997)
[43] Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.M.: Automatic Test Generation: A Use Case Driven Approach. IEEE Transactions on Software Engineering, 140–155 (2006)
[44] Prasanna, M., Sivanandam, S.N., Venkatesan, R., Sundarrajan, R.: A Survey on Automatic Test Case Generation. Academic Open Internet Journal 15 (2005)
[45] Coulter, A.C.: Graybox Software Testing Methodology: Embedded Software Testing Technique. In: Proceedings of the18th Digital Avionics Systems Conference (1999)
[46] Craig, R.: Measurement and Metrics for Test Managers. In: STAR East. SQE (2007)
[47] Kaner, C.: Measurement Issues and Software Testing (2001)
[48] Landis, L., Waligora, S., McGarry, F.: Recommended Approach to Software Development. Software Engineering Laboratory Series, pp. 81–305. NASA (1992)
[49] Bassin, K., Biyani, S., Santhanam, P.: Metrics to Evaluate Vendor-Developed Software Based on Test Case Execution Results. IBM Systems Journal 41(1), 13–30 (2002)
[50] Jorgensen, M., Shepperd, M.: A Systematic Review of Software Development Cost Estimation Studies. IEEE Transactions on Software Engineering 33(1), 33–53 (2007)
[51] Binkley, D.: Semantics Guided Regression Test Cost Reduction. IEEE Transactions on Software Engineering 23(8), 498–516 (1997)
[52] Leung, H.K.N., White, L.: Insights into Regression Testing [software testing]. In: Conference on Software Maintenance (1989)
[53] Nageswaran, S.: Test Effort Estimation Using Use Case Points. In: 14th International Internet & Software Quality Week (2001)
[54] Aranha, E., Borba, P.: An Estimation Model for Test Execution Effort. In: International Symposium on Empirical Software Engineering and Measurement, ESEM 2007 (2007)
[55] Abran, A., Bourque, P., Dupuis, R., Moore, J.W.: Guide to the Software Engineering Body of Knowledge - SWEBOK. In: Alain, A., et al. (eds.). IEEE Press, Los Alamitos (2004)
[56] Stocks, P.A., Carrington, D.A.: Test Templates: A Specification-Based Testing Framework. In: Proceedings of the 15th International Conference on Software Engineering. IEEE Computer Society Press, Los Alamitos (1993)
[57] Beizer, B.: Black-Box Testing: Techniques for Functional Testing of Software and Systems. John Wiley & Sons, Inc., Chichester (1995)
[58] Offutt, J., Abdurazik, A.: Generating tests from UML specifications. In: Proc. Second International Conference on the Unified Modeling Language (1999)
[59] Taylor, C.M.: EPDAV – A Model for Test Case Definition. In: Conference of the Association of Software Testing (2006)
[60] Bruno, M., Canfora, G., Di Penta, M., Esposito, G., Mazza, V.: Using Test Cases as Contract to Ensure Service Compliance Across Releases. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 87–100. Springer, Heidelberg (2005)
[61] Kaner, C., Falk, J.L., Nguyen, H.Q.: Testing Computer Software. John Wiley & Sons, Inc., New York (1999)
[62] Maletic, J.I., Soliman, K.S., Moreno, M.A., Mercer, W.M.: Identification of Test Cases from Business Requirements of Software Systems. In: American Conference on Information Systems AMCIS (1999)

[63] Baudry, B., Fleurey, F., Jezequel, J.M., Le Traon, Y.: Genes and Bacteria for Automatic Test Cases Optimization in the .NET Environment. In: Proceedings of the13th International Symposium on Software Reliability Engineering, ISSRE (2002)
[64] Menter, F.: CFD Best Practice Guidelines for CFD Code Validation for Reactor- Safety Applications. CFX, Germany (2002)
[65] Mikhailova, A., Doche, M., Butler, M.: Contracts for Scenario-Based Testing of Object-Oriented Programs (2002)
[66] Bruno, M., Canfora, G., Di Penta, M., Esposito, G., Mazza, V.: Using Test Cases as Contract to Ensure Service Compliance Across Releases. In: Benatallah, B., Casati, F., Traverso, P. (eds.) ICSOC 2005. LNCS, vol. 3826, pp. 87–100. Springer, Heidelberg (2005)