ABSTRACT
For nearly three decades the term "Test Case" (TC) has been widely used in software testing as a work unit, a metric and a documentation entity. A thorough review of the extant literature reveals that there is no formal and agreed-upon definition of a TC, in spite of its centrality and extensive use. Following Kaner's (2003) assertion that "good TC is one that gives the required information", we see benefits in formalizing a unified, well defined and structured TC format.

In this paper we present a brief literature review of the TC concept, explore the definitions of TCs, propose a classification of the various definitions into four categories, and highlight the conceptualization underlying each category. Undesired implications of this situation are then discussed, and four criteria for a 'good' TC definition are proposed while discussing benefits entailed by such a definition.

The paper proceeds by suggesting an alternative structural definition of a TC which complies with the above criteria. The proposed definition illustrates a TC as a composition of five structures. The first four are: TC factors, internal activities and flows, dynamic external interaction entities, and verification actions. A different, fifth structure is the TC output and results. We elaborate on the five structures, illustrate the definition and its benefits via the lens of the four criteria of a 'good' TC, and discuss its use in several applications showing that it addresses issues pertaining to various environments. In conclusion we discuss limitations and future research trajectories opened by the proposed TC definition, as well as its implications for practice.

General Terms
Design, Standardization, Theory, Verification.

Keywords
Software testing, Test case, Test case definition.

1. INTRODUCTION AND BACKGROUND
Software testing is a fundamental activity within the software development process that largely impacts the quality of the final product when bugs are discovered and corrected prior to the software deployment. It also consumes a substantial portion of the software development effort, between a quarter to a third of the time in common applications, and more in high-risk ones [1, 2].

The test case (TC) is the basic building block of most testing processes, thus a major part of the testing effort is dedicated to TC design, generation, and execution [3]. The TC is mentioned as a cornerstone of the testing process since the beginning of formal software testing in the late 1980s, for example in the System Test and Evaluation Process (STEP) framework [4]. Three sources for TC generation were identified: functional requirements, performance requirements, and the system's design [4]. A formal definition of a TC, however, was not included in this framework.

Since then TCs have been the focus of ample software testing research. For example, several works have discussed the issue of TC generation methodology. Ostrand & Balcer [5] suggested generating TCs as a collection of test frames and test scripts, yet these two terms were not precisely defined. Weyuker [6] maintained that TCs are formed by decision statements, recognizing that the more the number of decision statements in the tested code, the more complex is the TC. The centrality of the TC is evident in the work of Harrold, Gupta & Soffa [7], who used TCs as the basis of a methodology to minimize testing efforts, assuming that TCs stem from requirements. Others shared this view and developed the coverage matrix technique where TCs are scrutinized for requirement coverage in an effort to optimize testing efforts [8].

The growing complexity of software required new approaches to testing efforts, since TC generation and test results evaluation became more difficult [9]. Likewise, new platforms and development methods such as Service Oriented Architecture (SOA) and Test Driven Development (TDD) rendered the simplistic TC concept insufficient and called for more sophisticated approaches to TC identification, generation, and management [10-13]. It thus became evident that TCs are expensive to generate [14] and tedious to manage, hence should be appropriately maintained and stored for use and re-use [15].

In spite of the acknowledged variability among TCs size and complexity [16], they have been frequently used as simple metrics for estimating and controlling the testing effort. It is a common practice to use TC counting as evidence of the testing progress, assuming linearity in effort and execution time [17]. Likewise, variations and manipulations on number of TCs appear as basic metrics for testing assessment in several works [16, 18, 19]. This problematic approach, however, has been addressed by some researchers, albeit a handful. For example, Nageswaran [20] suggested using use case function points instead of counting TCs to assess testing effort, assuming that the complexity can be reflected by the assigned number of function points, and Aranha and Borba [21] advocated a similar approach by using execution points.

Evidently, TCs have been commonly treated as quantifiable and measureable entities, while in fact the literature does not provide a widely accepted definition that could serve as a foundation for this assumption. On the contrary, a thorough literature review in search for an agreed upon definition for a TC resulted in mixed
evidence. Of 267 papers reviewed that discussed software testing, only 38 (14%) included formal definitions of a TC, that could be classified into four dominant approaches or categories: 1) the input-process-output-objectives approach, 2) the states and transitions approach, 3) the contractual approach, and 4) other definitions. These four definition categories are summarized in Table 1.

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 objective. This is the approach adopted by IEEE STD 829.1998 [15]. 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. Notably, only the first two categories represent structured definitions that specifically imply at the components expected in a TC and their relationships (Wikipedia definition for 'structure'), whereas the other two are rather symbolic and unstructured. Of these two, contract is closer to the software engineering discipline, while definitions included in the last category generally stem from contexts alien to the software engineering world and are far from structured.

The lack of a formal TC definition poses several questions that merit further investigation. First, if TCs are not well defined, how can they be sensibly measured and quantified? Hence, how can they serve as sensible testing metrics? Second, how is their quality and adequacy assessed? Or in Kaner's [22] words, "what is a good test case?". Third, in light of the urgency to optimize and contain costs of software testing, how can TCs be automated when formal, structured definitions are either absent or vague?

In an attempt to answer these questions, characteristics of a desirable TC definition are brought next, followed by a detailed description of a newly proposed structural definition of a TC. The paper concludes with a discussion of the adequacy of the new definition and its potential benefits, as well as the need for further work.

A word of caution is however appropriate at this point. The present work adopts the engineering approach to software development, rather than the artistic one. The basic assumption underlying this work is that there is merit in executing software testing using methods and tools that are, to a large extent, structured, quantifiable, and measureable. Benefits, as well as shortcomings of this approach are discussed at the end of this paper.

2. A 'GOOD' TC DEFINITION

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

We maintain that a formal definition is indeed required. In fact, in real-world testing of life-threatening projects such as a nuclear reactor, 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 [23]. The recommendation is that each TC should be 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.

Thus, what should be the characteristics of a formal TC definition? It is suggested that a TC definition should possess four characteristics: 1) unambiguousness, 2) generalizability, 3) quantifiability, and 4) automatability. The following benefits stem from these characteristics: 1) Formal and unambiguously defined TCs would be uniformly understood by the various stakeholders participating in a testing endeavor. 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 Babylont dominating the testing world, and drive sharing expertise among various testing schools and perceptions. 2) An appropriate formal definition is one that is valid across platforms, testing domains, etc. thus allows broad re-use of TC structures. Generalizability ensures maintaining testing assets and investments along various testing efforts, namely, TC generation tools and techniques would be valid in different testing environments. 3) When defined in a quantifiable manner TCs can be sensibly measured, compared, and used as metrics. Currently, measurements involving counting TCs are clearly inconsistent. 4) There is no need to elaborate on the benefits rendered by automating TC generation, execution and management, a means believed to optimize testing efforts and contain costs when TCs are planned to be re-used. Several conditions are mandatory for developing general TC automation techniques, a formal definition of the TC structure being a fundamental requirement.

Examining the existing definitions through the lens of these characteristics illustrates the deficiencies in each type. The Input-Process-Output-Objective definitions are generally unambiguous, but not necessarily measureable and quantifiable. For example, 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 interactions 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 origin 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 [10] defined a test as a contract between the user and the software provider, whereas Mikhailova et al. [24] defined testing as a contract between the system under test and its environment. Clearly, only a formal definition of the contract, such as the one attempted by Aichernig [10] 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 suggest 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 a formal definition of TCs, hence its absence is possibly one of the barriers to a broader diffusion of automation tools even in cases automation is clearly feasible. These shortcomings are quite likely among the causes for the annual economic damage equivalent to $20 – $52 billion 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 clearly warranted.

Table 1: TC definitions

<table>
<thead>
<tr>
<th>Category</th>
<th>Definition</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input-Process-Output-Objectives</td>
<td>&quot;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&quot;</td>
<td><a href="http://www.wikipedia.org">www.wikipedia.org</a></td>
</tr>
<tr>
<td></td>
<td>&quot;A test case is the combination of test data and oracle information to determine the validity of the test&quot;</td>
<td>[25, p. 9]</td>
</tr>
<tr>
<td></td>
<td>&quot;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&quot;</td>
<td>[15, p. 187]</td>
</tr>
<tr>
<td></td>
<td>&quot;Test case is a test vector consisting of a set of test inputs and the corresponding test outputs (pre and post conditional assertions)&quot;</td>
<td>[26, p. 2]</td>
</tr>
<tr>
<td></td>
<td>&quot;Test Case is an identified set of information including inputs and expected outputs associated with a particular program behavior&quot;</td>
<td>[27, p. 7]</td>
</tr>
<tr>
<td></td>
<td>&quot;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&quot;</td>
<td>[28, p. 2]</td>
</tr>
<tr>
<td>States and Transitions</td>
<td>&quot;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&quot;.</td>
<td>[29, p. 13]</td>
</tr>
<tr>
<td></td>
<td>&quot;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.”</td>
<td>[30, p. 47]</td>
</tr>
<tr>
<td></td>
<td>&quot;Test case is composed of several components: test case values, prefix values, verify values, exit commands and expected outputs&quot;</td>
<td>[31, p. 28]</td>
</tr>
<tr>
<td></td>
<td>&quot;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 Which may require preconditions, Pp All of which runs in environment, Ec Hence, a Test Case, Tt = Ec Pp Dd Aa Vv&quot;</td>
<td>[32, p. 51]</td>
</tr>
<tr>
<td>Contract</td>
<td>&quot;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&quot;.</td>
<td>[10, p. 8]</td>
</tr>
<tr>
<td></td>
<td>&quot;a form of contract between a service provider and a service user”</td>
<td>[33, p. 2]</td>
</tr>
<tr>
<td>Other</td>
<td>&quot;An empirical frame of reference, rather than a theoretical one”</td>
<td>[34, p. 359]</td>
</tr>
<tr>
<td></td>
<td>“…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”</td>
<td>[22, p. 2]</td>
</tr>
<tr>
<td></td>
<td>&quot;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&quot;</td>
<td>[22, p. 2]</td>
</tr>
<tr>
<td></td>
<td>&quot;a specific set of attribute values that tests a given logical situation”</td>
<td>[35, p. 3]</td>
</tr>
<tr>
<td></td>
<td>“a test case can be considered as a predator while a mutant program is analogous to a prey”</td>
<td>[36]</td>
</tr>
</tbody>
</table>
3. A NEW TC DEFINITION

The following schema illustrated in Figure 1 presents a test case definition as a composite of five separate interacting structures or components, each of which is composed of internal elements or items. Each element in a structure has its own properties and may or may not participate in the actual execution of a specific test case. Hence, a component may be empty thereby not a part of a particular TC when its functionality is outside the scope of the specific TC objective.

Figure 1. TC structural definition

As illustrated in Figure 1, a TC is composed of five main components: (1) Test case factors – all preliminary properties required to be set up prior to the execution of the TC, (2) Test case internal activities (TIAF) – all entities participating in the actual flow and execution of the TC which define the core execution path tested by the TC, (3) Dynamic external interaction entities (DEIE) – mostly data items and results of external interactions with the main flow of the test case, (4) Basic verification action (BVA) – external operational entities that observe selective data items determining the validity of their value at a specific moment. The fifth structure, TC outputs and results (5) is treated somewhat differently than the other four components, since it represents the final results, or the overall outcome of the TC operation. All the above TC items should possess the following characteristics:

- **Measurable** – items should be measurable in order to allow quantifiable metrics to calculate workload, effort, time boxes, etc. Measurability should obey the definitions suggested by Kaner and Bond [37]. Measurability of DEIE and BVA items refers to item units rather than the entities they interact with, since they represent results of interactions with entities external to the effort required to build the actual TCs. The effort of building the external interaction entities is part of preparing the test environment which is outside the scope of the TC measurement. This comment also pertains to the test factors which might relate to externalities not calculated as part of the TC as well.

- **Reproducible** – in order to be able to reproduce a certain event, it is mandatory that certain conditions can be reproduced.

- **Traceable** - the origin and reasoning of the value of each item should be understood and known.

- **Documentable** – an item's value and properties can be documented in case of need to analyze its effects on the test outcome. This property might be considered an outcome of the two previous ones.

- **Repeatable** – items should be repeatable in order to ensure that intended tests can be repeated when required.

3.1 Test Case Factors

Test factors define the initial environmental state prior to the TC execution.

**TC factor definition:** A TC factor is any preset variable whose values are to be controlled during the tests. Examples are hardware and operating system configurations, load characteristics, software versions, operating modes, input data, and so forth [38]. In fact, it is quite difficult to precisely define a test factor yet experienced testers know what a test factor is.\footnote{We thank Javier Tuya for this comment}

**TC factor purpose:** To enable the test to start from a pre-defined initial state.

**TC factor values:** A TC factor value is a specific value assigned to one of the factors during the test. For example, a browser test factor value could be IE 7.0 or Firefox 3.0. Display resolution values could be low, medium or high.

**TC Factor Requirements**
- **Stable** – by definition, a factor ought to be stable during execution, ensuring that no nondeterministic shifts occur throughout an execution cycle.
- **Represent customer setup** – from a practical point of view TC factor values should be closely related to a customer system setup.

**TC Factor Classes (types, origins)**

Following is an exemplary yet inconclusive list of TC factor classes:

- **Hardware** – for example: a specific machine, CPU, file system, storage etc.
- **Operating system** – similar to hardware is the type of operating systems and other system characteristics such as file system, databases etc.
- **Software configuration** such as: load characteristics, software versions, operating modes, security flags, runtime parameters, etc.
- **Preliminary input data items** such as: application setup parameters, application data set requirements etc.
- **Preliminary operational running jobs** – in order to start a test from specific application state – a set of executables may be needed to function in the background, such as – servers, demons, processes, network elements etc.

3.2 TC Internal Activities & Flow Items (TIAF)

**TIAF definition:** A list of activities and their relationships, termed as flows, which describe the executable path tested by the specific TC. For example, a TIAF can be described by three...
activities (items) that are related by a branch choice (flow item) together composing the TIAF vector. TIAF can relate to black-box testing, causing the items to collapse into just a few or even one entity, or to gray-box or white-box testing, resulting in detailed list of items.

**TIAF purpose:** To describe the actual flow of the test case.

**TIAF item values:** A common representation of this vector of activities and flows is state transitions [39]. Another can be a more loosely bound yet deterministic flow of events operated by a certain algorithm for the testing flow. For example Ding et al [40] suggested the following optional elements for a test case flow: Sequential, Branch and Choice, Loop, and Parallel, which are collected by using several program slicing algorithms [41].

Regardless of the approach adopted, activity and flow items identify atomic elements, such that further decomposing them is senseless, which together form the actual test case execution flow.

**TIAF item requirements:**

In addition to the common characteristics described above, a TIAF item should also possess the following:

*Single access and/or exit point* – A TIAF item should have a single access and a single exit points in order to ensure the atomic property of the element, which means that further decomposition of the items is senseless. Although it might be sometimes challenging to identify these single points, we believe it is possible in several cases.

**TIAF Activity item Classes:**

An inconclusive, exemplary list of activity items is as follows:

- **Dynamic parameter set up action** when parameters need to be modified during execution
- **Execution of a job** – This is the most common type of activity. It can vary from calling a service to execution of a single command.
- **Script manipulation** – by calling a script a sequence of events can be applied upon a single action. This activity should be cautiously used since it may result in losing control during the script execution.
- **API activation** - either internal or external
- **Status report** – an intermediate output report can be included as part of the flow, enabling the control mechanism to present the current position and status of the TC during its execution.
- **Verification call** – this item is discussed in chapter 5.1.4 usually being called as part of the execution flow.
- **External interface call** – a simple call to an interfacing entity.

As stated above, each TIAF item is counted as one entity regardless of its type.

**TIAF Flow Item Classes:**

As stated above, classical basic flow items describe activity flow such as sequential, parallel, branching etc. However, more complex flows may be used. The following are examples of such flows:

- **Flows that are determined by application components** – often an application flow is described within the tested application, determining the course (sequence) of actions to be performed during the test case execution, rather than explicitly as a flow item.
- **Testing shortcut** sometimes an artificial flow is needed (when software development is not finished) bypassing the unfinished areas.
- **Instrument element** (for control and documentation) – a common practice during testing is to instrument the flow by printing (or other output channel) notifications to ease the flow tracking.

### 3.3 Dynamic External Interaction Entities (DEIE)

**DEIE definition:** A component representing the results of interacting with external entities. Examples of DEIE are data returned by a service called by the TC [42], or the location of a piece of data. In contrast to test factors that are steady throughout the test process, DEIE are dynamic interactions occurring as a part of the test flow.

**DEIE purpose:** TC execution may often require interaction with external input/output, or execution of external entities such as services or other operations to be performed. These entities will usually be pre-planned but be dynamically operated.

**DEIE values:** The DEIE can be described as a result of an external interaction with a data injection mechanism, a stub, or by another mechanism enabling the TC to complete its task regardless of a missing complete testing environment.

**DEIE requirements:**

Relate to a specific TIAF – each DEIE should be related to a specific TIAF in order to ensure singularity of behavior. This means that results can be uniquely predicted each time the TC is executed.

**Atomic** – DEIE items are treated as atomic regardless of their actual structure, ensuring that no further testing is required during the TC execution, since further testing, for example of external stubs, is beyond the scope of the particular TC. In fact, the atomic DEIE item is the result of the interaction, rather than the interaction or the external interacted item. Therefore, each DEIE can be counted as one item, and all DEIE items can be equally weighted.

**DEIE Item Classes**

Examples of DEIE items are: Single/Multi Data Item, Memory resident data or data address, Memory temporary flag, Data retrieved from an external API, etc.

The above TC components compose the TC structure, yet are insufficient in describing the TC goals. This and other details are part of the TC documentation, an issue outside the scope of this work.

### 3.4 Basic Verification Action (BVA) Items

**BVA item definition:** A BVA item is an external investigator entity that observes and documents selective states and signals their validity at a specific time [43].

**BVA purpose:** To anticipate, control and document certain behaviors during a TC execution, in order to determine the exact location and time of the occurrence of a fault. When applied to
the TC final output, the BVA is a logical representation of the test oracle.

**BVA item values**: A BVA item is composed of expected input data items, arithmetic and logical operators to be executed on the input data to determine its correctness, and a result distribution method, as illustrated in Figure 2.

![Figure 2. BVA flow](image)

**BVA Item Requirements:**

Accesses all types of storages and data items – in light of the growing complexity of modern applications and the data units involved, BVA items should be able to access all types of data and storage types. For example, data can be represented by simultaneously using various techniques as databases, XML files, encryption, compression, coding, and dynamic data location [43].

External to the tested application – BVA items should be stored, maintained, managed, controlled and operated as external entities separated from the actual TC data, preferably as API calls.

Interfacing with at least one reporting system – to facilitate the result distribution mechanism.

Fast / Efficient – verification actions may frequently appear during the TC execution thus should not hinder the execution performance.

Able to handle complex logic and data structures – the actual verification may be the result of relatively complex logic and multiple accesses to data items, including performing complex calculations in order to validate the correctness of the data items. For example, a BVA item may examine the correctness of a certain calculation such as rate amount, or the relations between two numbers.

**3.5 TC Output and Results**

The TC output and results are the outcome of applying a set of BVA items on the output issued by the application under test. This action is required because determining the correct outputs of tested applications presents a real challenge since it can be simultaneously represented in various forms such as databases, XML files, encryption, compression, coding, dynamic data location, etc. Consequently, a deeper understanding of the results structure and characteristics is required, by using sophisticated tools and processes, termed oracle, to determine the test outcome. Thus, the TC outputs and results are in fact the results of the operation of a vector of BVA items on the application output, forming the logical representation of this oracle.

**4. DISCUSSION**

In this work a new TC structural definition is proposed, where a TC is composed of four distinctive components, each of which possesses characteristics allowing more precise identification, quantification, and documentation of TCs. It also presents a new approach to validating the TC course of execution, as well as its outputs and results, by using a vector of verification actions which, when applied on the output, represents the oracle. Although some effort in this direction might be seen in specific domains, for example testing Java code (JUnit tools by Sun Microsystems), this work takes this effort a step forward by 1) suggesting a formal definition, and 2) extending some of these concepts to the general testing domains. Thus, the proposed TC definition and structure applies not only to Java code but also to functional, business, and non-functional testing.

The proposed structure also applies to the two common approaches to describing TC behavior: the state machine approach – where each step taken during TC execution is considered a transition between states [44], and the stateless approach where the actual flow of the TC is less rigid and could be described in a flow chart [45]. However, whether a TC is structured as the former or the latter can be considered a property of the TC regardless of the system under test. The two approaches are illustrated next.

**4.1 The TC as State Machine**

The TC as a sequence of transitions between states was adopted by Microsoft for Model Based Testing (MBT) [44], although a formal definition for a TC could not be found. Figure 3 presents the way such a model is implemented by the new approach. When the states & transitions model is applied all relevant verification action calls are being called at a specific stage – checking each state following the previous transition. It is thus notable that the TC output can be actually merged with the last state reached by the test case. This approach enables the breakdown of multi state test case into smaller cases where the previous state serves as the test factor for the next state, simplifying the TC schema to the one illustrated in Figure 3:

![Figure 3. TC as a state machine](image)

**4.2 The TC as a Stateless Machine**

The stateless approach (Figure 4) is taken when creating rather complex test cases. For example testing a multi layer service, where a service is calling another service as part of the internal flow [42], or when viewing TCs using the gray box approach [45, 46]. An important advantage of this approach lies in the role of the verification calls during the execution process, which can
identify a failure long before the test case has been completed, saving expensive testing time.

Applying a vector of verifications to the final output serves as the final check of the test results.

4.3 The New TC Definition and the Four Good TC Criteria

As stated above and in line with the engineering approach adopted in this work, a good TC should possess as a minimum four characteristics: unambiguosity, quantifiability, generalizability, and automatibility. The present newly proposed definition adheres to these attributes.

Unambiguosity: for a TC to be unambiguously defined its entry parameters, internal flow, external interactions and expected results should be explicitly stated. When so defined, it is quite likely that most testers will similarly interpret the TC. Clearly, the proposed structural definition renders the TC unambiguous. Clearly, TCs will still differ in their objectives, thus some TCs will be clearer to technology professionals, while others – to business professionals. This, however, does not imply ambiguity in the sense intended here. Some of the existing automatic tools, particularly for code testing, already produce TCs that are unambiguous, yet these are specific to testing code in a unique programming language, hence do not obey the 'generalizability' quality, as discussed later.

Quantifiability: a TC is quantifiable when it can be objectively measured and evaluated using agreed upon units. The proposed TC definition is composed of sets of items, each of which is quantifiable, rendering the whole TC structure quantifiable.

Generalizability: the proposed definition is based on general concepts, not affiliated with a specific platform, domain or environment.

Automatibility: the above three attributes of the proposed definition are mandatory for TC automatic generation and execution. Furthermore, since the newly proposed TC is composed of distinctive structures of items, it can be documented, stored and maintained for re-use by efficient automation tools. Thus, automatibility refers to automatic TC generation, execution and maintenance.

4.4 The Proposed TC Definition and TC Definition Categories

As illustrated in Table 1 above, TCs are generally defined under four main categories: input-process-output-objectives, states and transitions, contract, and others. Thus, does the new definition create a fifth category? We maintain that the answer is negative, since the proposed definition contains the first three of the four categories. Clearly, the first and second definition categories are well represented by the TIAF component of the TC, and by the outputs and results interpreted by the BVA vector which represents the oracle. A more sophisticated alignment is proposed between the verification items and the contract definition category. It is proposed that the BVA items can be easily converted into contract items, yet this conversion is beyond the scope of this work.

Evidently, the proposed structure cannot support symbolic definitions such as ‘predator and prey’, since the engineering approach adopted cannot easily represent metaphors and symbols. Rather, clear and definitive testing objectives and TCs are assumed as the domain covered by this work. For example, a question such as "what happens to the system when event X happens" is not well supported by TCs formulated based on our structured definition. This type of testing should be differently dealt with.

5. CONCLUSIONS

This work proposes a new, structural approach to TC definition stemming from the need to make testing more efficient. Furthermore, structurally defined TCs can perhaps facilitate better completeness and consistency checks, yet this should be further investigated in future work. Nonetheless, as in every work, several limitations should be noted.

5.1 Limitations

First, as is the case with all new endeavors, this study merits further research and elaboration. The suggested structure and definitions should be subject to theoretical and empirical elaboration and validation.

Second, it should also be noted that the suggested work refers only to the TC itself in isolation from the testing environment and platforms, such as operating systems, data, stubs and simulations required for the actual test execution. This is important when referring to the measurability and quantifiability attributes of the various items and components.

Finally, the structural TC approach overlooks the soft and flexible modern attitude towards software testing, and the unique skills and special position of the testing experts. Approaches such as exploratory testing may perhaps co-exist alongside the proposed structural approach where rather ‘artistic’ conceptualization is appropriate. For example test cases like: "It should take < 4 seconds to compute the result; preferably < 2", or "The cancel button should NOT suddenly grey itself out", or

2 We thank Scott M. Allman for this comment.
"The number four should appear in BLACK, not RED\(^3\), makes sense to a freehand tester but can be unfeasibly costly when attempting a translation to the new structural model.

5.2 Future Enhancements

If supported by further research and practice, this work opens new trajectories for further enhancements of the proposed structure.

First, it is suggested that the TIAF can be extended to map unto business entities by replacing TIAF with BIAF (business internal activities & flow), enabling the description of the TC internal flow in a formal business terminology. Introducing BIAF may close a gap between the technical aspects software usage and its actual business representation, since a single flag may now signal the business implication of the input. Thus, it would be possible for business users to define the TC TIAF (or BIAF), and it would also bring the TC notion closer to the 'contract' definition category. This, however, should be further investigated since it is necessary to still maintain the characteristics of measurability and quantifiability.

Second, it might be possible to consider separation of testing items such as DEIE, BVA, as distinctive inventory DBs that are generated, maintained, and re-used. If indeed possible, this would clearly ease testing management. Third, the proposed structure might facilitate re-thinking the way test cases are built; build the DEIE and BVA, as well as other TC structure inventories prior to generating the actual TCs, in collaboration with the software developers during the design process. For example, quite often the external interactions and verifications are known during the design phase, and can therefore be generated and stored before the actual testing phase is reached. This concept is particularly attractive when using agile development processes. Efforts in this direction are attempted by Java code testing tools such as GroboUtils, yet, as stated earlier, these are specific rather than general tools.

Fourth, since the proposed definition is structured, it allows designing test cases with automation intention and mechanism from the outset, facilitating actual automation when available and feasible.

Fifth, the proposed structural definition suggests development of new testing tool-set to support the new model. Finally, it is suggested that empirical studies define measurements and examine the improvement in testing processes entailed by using TCs that are structured according to the proposed definition.

\(^3\) We thank Matt Hauser for this comment and examples
6. REFERENCES


