TePIA: A taxonomy of test case prioritisation information attributes

— *Extended analysis* —

Aurora Ramírez[1], Robert Feldt[2] and José Raúl Romero[1]

[1] Dept. Computer Science and Numerical Analysis, University of Córdoba

[2] Software Engineering Division, Chalmers University

December 9, 2020

1 Introduction

The TePIA taxonomy formalises the definition of information attributes for test case prioritisation (TCP). It is organised into dimensions, which are further decomposed into categories. One or more values per category are assigned to describe the identified attributes. The taxonomy thus gives a structured way to describe and group information attributes. Figure 1 shows the dimensions and categories of the TePIA taxonomy, which are described next, together with their possible values. Then we present the actual attributes we found in the literature based on how they map into the taxonomy.

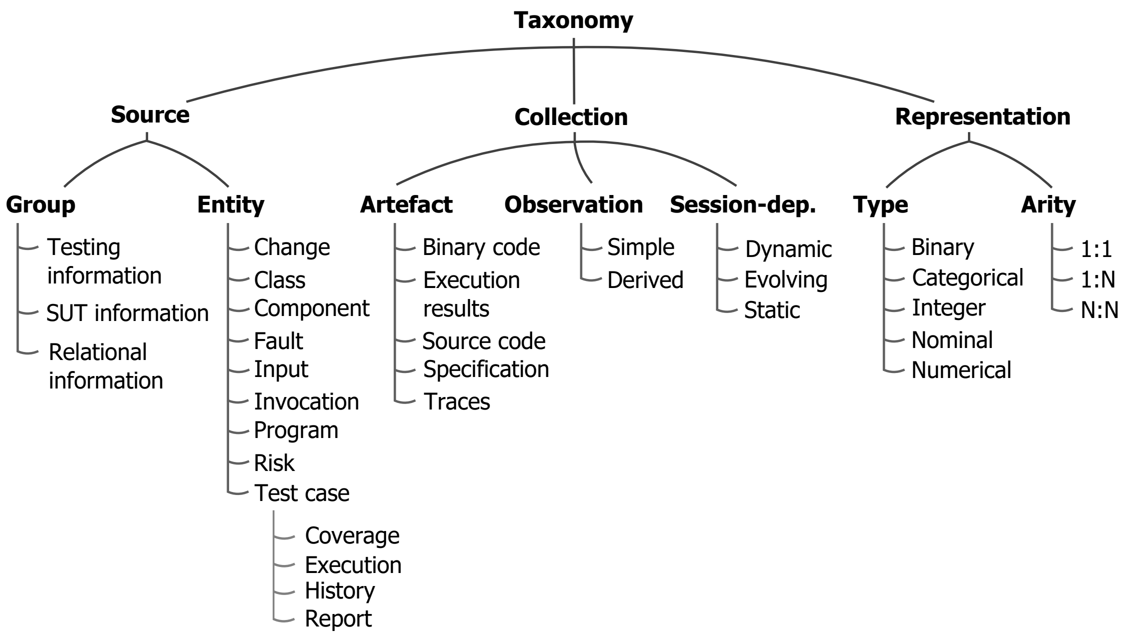

Figure 1: Taxonomy of information attributes for test case prioritisation.

1. **Source**. This dimension specifies where the information is generated or originates. It is further decomposed into the following categories:

   (a) *Group.* A high-level view of the nature of the information, divided into three categories. Firstly, *testing information* attributes only require the analysis of test cases or its outcomes. Secondly, *SUT information* attributes are based on characteristics of the system under test (SUT). A third category, referred as *relational information*, lies in between with the aim of grouping those attributes that need to establish connections between test cases and the SUT, e.g. which functionalities are covered by each test case.

   (b) *Entity.* An abstract representation of the type of element from which the information is going to be extracted. The element can be a part of the system at different levels of abstraction (program, component or class). It can also represent an action generating information about the SUT, such as an invocation or change, or something derived from the specification, e.g. risk and inputs. Focusing on the testing activity, test cases and the faults they expose are important entities too. Test case, as a primary entity of the TCP process, is further decomposed to analyse the different aspects considered so far.

2. **Collection**. This dimension serves to analyse how the information attribute is obtained from the entity. The following categories are defined:
   (a) *Artefact.* The artefact that contains the entity and should be parsed or instrumented to obtain the attribute value. The possible options are: binary code, execution results, source code, test case/system specification, and traces.
   (b) *Observation.* A category to specify whether the attribute value is directly obtained as a raw value from the artefact or if it is derived from other measures.
   (c) *Session-dependent.* This category specifies whether the attribute value changes when a new testing session is carried out (dynamic) or not (static). A third option, named "evolving", indicates that changes in the SUT might affect testing outcomes.

3. **Representation**. This dimension seeks to describe how the information attribute is computationally stored after processing the collected data. Two categories are defined:
   (a) *Type.* The sort of computational variable that is assigned to the attribute. The nomenclature by Pyle [1] is considered: binary, categorical, integer, nominal —including text— and numerical (floating-point number).
   (b) *Arity.* Represented as a tuple N:N, it describes whether several entities are involved in data computation. The first element specifies if the attribute is computed for a single entity (e.g. a test case) or a group of entities (e.g. a test suite). The second element indicates whether the value depends on a single entity (e.g. the test case itself) or many (e.g. other test cases o several pieces of code). In short, this category will allow establishing one-to-one, one-to-many and many-to-many relations among the values needed to compute the attribute.

2 Dimension 1: Source

The aim of the first dimension is to describe the information available for TCP by identifying where the information come from and which entities are involved. This first categorisation allows researchers and practitioners to find information attributes related to the sources they can access to as part of the testing process. The analysis is structured based on the *group* category, as explained next while the accompanying tables detail how the attributes map to the other dimensions and categories of the taxonomy. For each information attribute, we include the references to the papers defining or using it. For those attributes appearing in many studies, the full list of supporting references can be found in the GitHub repository.

2.1 Testing information

Table 1 shows the attributes belonging to this group. These attributes are expected to be highly applicable to any TCP technique, since the test case is the unique entity in

Table 1: Classification of attributes based on testing information.

Source		**Collection**			**Representation**	
Entity	*Attribute*	*Artefact*	*Observation*	*Session*	*Type*	*Arity*
Test case dependency	Implementation dependencies	Spec.	Derived	Static	Int.	1:N
	Joint execution	Spec.	Simple	Dynamic	Cat.	N:N
	Verdict pattern	Exec.	Derived	Dynamic	Nom.	1:N
Test case execution	Allocation time	Exec.	Simple	Static	Num.	1:1
	Cost	Spec.	Simple	Static	Cat.	1:1
	Execution time	Exec.	Simple	Dynamic	Num.	1:1
	Resource utilisation	Traces	Derived	Dynamic	Num.	1:N
	Total time	Spec.	Simple	Static	Cat.	1:1
Test case history	Historical effectiveness	Exec.	Derived	Dynamic	Num.	1:N
	Historical executions	Exec.	Derived	Dynamic	Int.	1:N
	Historical fails	Exec.	Simple	Dynamic	Int.	1:N
	Historical verdicts	Exec.	Simple	Dynamic	Bin.	1:N
	Previous execution	Exec.	Simple	Dynamic	Bin.	1:1
	Previous priority	Exec.	Simple	Dynamic	Int.	1:1
Test case property	Age	Exec.	Simple	Evolving	Num.	1:1
	Estimated fault detection	Spec.	Simple	Static	Cat.	1:1
	Resources	Spec.	Simple	Static	Bin.	1:N
	Size	SC	Simple	Static	Num.	1:1
	Static priority	Spec.	Simple	Static	Num.	1:1
	Status	Spec.	Simple	Dynamic	Bin.	1:1
	Textual description	Spec.	Derived	Static	Num.	1:N
	Type of test	Spec.	Simple	Static	Bin.	1:1
Test case report	Effectiveness	Exec.	Simple	Dynamic	Bin.	1:1
	Failure frequency	Exec.	Simple	Dynamic	Num.	1:N
Test case similarity	Code similarity	SC/Tra.	Derived	Static	Num.	N:N
	Input similarity	Spec.	Derived	Static	Num.	N:N
	Text similarity	Spec.	Derived	Static	Num.	N:N

Source: BC=Binary code, Exec=Execution results, SC=Source code, Spec=Specification, Tra=Traces
Type: Bin=Binary, Cat=Categorical, Int=Integer, Nom=Nominal, Num=Numerical

any testing activity. Despite that only one entity, the test case, has been identified for this group, a wide variety of aspects can be measured. Firstly, a number of properties directly related to the test case definition are its *age* —time since its creation—, its *size* —usually measured as lines of code—, its *type* —e.g. acceptance vs. functional test—, its *status* —whether it is new or modified— or its *fault-detection* capability (as estimated by a tester). We note that the *type* attribute might include other types of test cases meaningful in other contexts, such as system, integration or performance tests. Similarly, the *status* of the test case could need a more fine-grained classification. Based on this information, TCP techniques give more priority to either newly developed test cases [2] or those supposed to find more faults [3]. The *textual description* of the test case, mapped to the frequency of words appearing in the test case specification, and the *resources* required to run the test case, are other possible attributes. With this information, the TCP technique might opt for test cases linked to certain functionalities (represented by the words) [4] and take the distribution of resources into account [5], respectively. Dependencies and similarities between test cases are often considered to make informed decisions during TCP. On the one hand, identifying dependencies can reduce the need of ordering all test cases under different assumptions: (*a*) the tester knows that a pair of test cases should be considered together (*join execution*) [6]; (*b*) the test case has *implementation dependencies* with other test cases [7]; or (*c*) the TCP method discovers *verdict patterns* describing interrelated outcomes [8]. On the other hand, test case similarity is defined as the distance between test cases based on a specific criterion. The underlying idea is that choosing diverse test cases somehow guarantees that different functionalities are being tested. Here, current TCP techniques have considered *code*, *inputs* and *text* (comments, identifiers and literals) similarity scores.

The rest of the attributes refer to properties of the test cases collected during or after their execution. From test case execution, TCP techniques might use attributes such as *allocation time* —time needed to prepare or configure the test case—, estimated *cost* of implementation and setup, *execution time*, *resource utilisation* (CPU, memory and I/O required by the test case), and *total time*, i.e. time needed to implement, configure and run. Time-aware TCP is a recurrent topic due to the necessary trade-off between duration and effectiveness when not all test cases can be run [9, 10]. Once a test session finishes, the test case *effectiveness* —whether it passes or fails— and the *failure frequency* —ratio of fail verdicts— can be measured to update prioritisation models [11, 12]. Outcomes from previous executions, known as test case history, allow including a long-term view of TCP performance. The following attributes have currently been applied: *historical executions*, which counts the number of times a test case has been executed; *historical effectiveness*, defined as the ratio between test case fails and runs; *historical fails*, which refers to the number of sessions for which the test case has failed; *historical verdicts*, i.e. the sequence of results in the last n sessions, *previous execution*, which indicates whether the test case has been executed in the previous section; and *previous priority*, which refers to the ordering position of the test case in the previous session.

2.2 SUT information

Table 2 shows the list of attributes within this group, which serve to adapt the TCP technique to the particularities of the tested system. Artefacts at different levels of abstraction, and aspects regarding their invocation and evolution, constitute the information entities here. Among the code artefacts for which an attribute has been computed, classes constitute the finest-grained entity with six metrics currently used in the literature. The six metrics defined by Chidamber and Kemerer [13] have appeared in TCP studies, used alone or in combination: CBO (coupling between objects), DIT (depth of inheritance tree), LCOM (lack of cohesion in methods), NOC (number of children), RFC (response for a class) and WMC (weighted methods per class). Similarly to WMC, weighted attributes per class has appeared as metric too. Other metrics defined at the class level are focused on class methods, including the number of *invocations* and the number of *inherited* or *overridden* methods. The *class size* —in terms of lines of code— and an estimation of its *fault-proneness* complete the list of class metrics. In general, all these metrics measure the complexity and level of dependency of classes, guiding the selection of those fault-prone modules that should be tested first [14, 15].

Similarly, component *instability* —a ratio between required and provided components— has been used for TCP at a higher level of abstraction [16]. The rest of artefact-oriented metrics refer to the whole program studied at different granularity levels. For instance, program *cohesion* has been defined based on package, component, class and method analysis [17]. Based on expert's judgement, one TCP technique incorporates the *type* of system

Table 2: Classification of attributes based on the system under test.

Source		**Collection**			**Representation**	
Entity	*Attribute*	*Artefact*	*Observation*	*Session*	*Type*	*Arity*
Change	Buggy change	SC	Derived	Evolving	Bin.	1:N
	Change intensity	BC	Derived	Evolving	Num.	N:N
Class	CBO	SC	Derived	Evolving	Num.	1:N
	Class size	SC	Simple	Evolving	Int.	1:1
	DIT	SC	Simple	Evolving	Int.	1:N
	Fault-proneness	BC	Derived	Evolving	Num.	1:N
	Invocations	BC	Derived	Evolving	Num.	1:N
	LCOM	SC	Derived	Evolving	Num.	1:N
	NMI	SC	Simple	Evolving	Int.	1:1
	NMO	SC	Simple	Evolving	Int.	1:1
	NOC	SC	Simple	Evolving	Int.	1:N
	PIM	SC	Simple	Evolving	Num.	1:1
	RFC	SC	Derived	Evolving	Num.	1:N
	WAC	SC	Derived	Evolving	Num.	1:1
	WMC	SC	Derived	Evolving	Num.	1:1
Component	Instability	Spec.	Derived	Evolving	Num.	1:N
Inputs	Data patterns	BC	Derived	Static	Num.	N:N
	Parameter values	Spec.	Simple	Static	Nom./Num.	1:1
Program	Cohesion	SC	Derived	Evolving	Num.	1:1
	Complexity	SC	Derived	Evolving	Num.	1:1
	Frequency of use	Spec.	Simple	Static	Cat.	1:1
	Type of system	Spec.	Simple	Static	Cat.	1:1
	Version	Spec.	Simple	Static	Int.	1:1

Source: BC=Binary code, Exec=Execution results, Spec=Specification, SC=Source code, Tra=Traces
Type: Bin=Binary, Cat=Categorical, Int=Integer, Nom=Nominal, Num=Numerical

(with respect to its installation procedure) and the *frequency of use* (linked to the importance of different parts of the system) in its decision process [18]. Similarly to the type of test case, these attributes could include additional values than those originally proposed by the authors.

Two other entities are not mapped to code artefacts, but they also provide SUT-related information. On the one hand, program inputs, expressed in form of either *data patterns* or *parameter values*, have been analysed to induce those values that make test cases fail [19]. On the other hand, *buggy change* estimations [20] and the *change intensity* [21] —semantic similarity between program versions— can be included in TCP models to guide the process towards recent SUT modifications.

2.3 Relational information

This group presents not only the largest list of attributes, but also the broader range of entities, as shown in Table 3. This fact confirms that many TCP methods need to establish connections between test cases and the SUT, but that these connections might come from very different places. As expected, test case coverage is the most popular relational information asset for TCP in the literature [22, 23], for which up to 16 different attributes have been identified. *Functional coverage* for object-oriented systems has been defined at multiple levels, the most common being statement, block and method. Other formulations take configuration [24], changed methods [25], database entities [26], GUI steps [27], inputs [28] or conditions, a.k.a *MC/DC coverage* [29], as the elements to be covered. It is even possible to find a *user-defined coverage* function [3]. Computing coverage only for the test cases not selected yet, i.e. the "additional approach" [30] has been applied to functional, MC/DC and change coverage. Other attributes use coverage information to identify and quantify the parts of the SUT that are more relevant to each test case. Within this group, we find *coverage distance* [31], i.e. distance of test cases based on their coverage; *coverage frequency* [32], which counts the number of times that each code block is covered; *coverage percentage* [21], which calculates how much of a block is covered (in percentage); *coverage profile* [33], that retrieves the groups of statements and branches covered by the test case; and *historical coverage* [11], which considers how frequently each function has been covered in previous testing sessions. Finally, coverage information can be combined with other properties. Some examples are the *complexity coverage* [34], which computes the cyclomatic complexity for only the functions covered by the test case; and the *weighted coverage* [35], obtained as a ratio between a coverage measure and the test case size.

Similarly, the analysis of calls from test code to system code constitutes another way to establish the connection between test cases and the functionality under test. In contrast to coverage attributes, those within this category (invocation entity) do not need access to SUT code lines, since they inspect how test cases make invocations. In particular, we found attributes that indicate when and how often a test case executes a code block (*active blocks* attribute) [36], the number of invocations to changed methods (*change calls*) [37], the number of invocations linked to previous fails (*fail calls*) [38] or the *sequence calls* executed by a test case [39]. Related to the type of invocations, it is possible to prioritise test cases based on the number of *method calls* (as a surrogate of method coverage) [40] or to give more importance to certain statements identified by the tester (*relevant statements*) [41]. Next entity in Table 3 is the program, for which four attributes were found. The *inputs* that the test case passes to the system and the *outputs* received are used to cluster similar test cases prior to prioritisation [42]. Also, the name and number of functions associated to system tests (*system function*) [43] have been applied to TCP to estimate reliability of a safety-critical system, whereas the similarity between *usage patterns* [44] allows including the impact of faults on different users in the TCP process.

The last three entities, namely risk, fault and change, are more related to the project specification and evolution. When risks are properly identified and documented, it would be possible to prioritise test cases based on its *risk coverage* [45] or to sort them according to the criticality of the components that each test covers (*component risk*) [16]. Test cases can also be ordered with respect to their ability to detect risky faults (*risk exposure*) [46]. However, this attribute assumes that faults have been related to risks beforehand. Fault-related information is widely studied for TCP according to our literature analysis. Among others, fault *age*, its *probability* of occurrence and its *severity* should be mentioned [47, 4]. These properties are used to compute the average age of the faults detected by the test case, its probability of finding undetected faults, and the sum of priorities assigned to the detected faults, respectively. The number (or ratio) of exposed faults (*fault count*) [4] and *mutants killed* [30] are other indicators of the test case detection capability. An estimator of the fault-proneness of the functions covered by the test case has been developed in this context too [22]. Focusing on change-based information, the content and priority of requests and issues have been studied as drivers of the TCP process. More specifically, the *issue score* attribute assigns a priority to those issues addressed in previous testing sessions, and the *failing issues* attribute considers the frequency of words appearing in issues related to failing test cases [48]. Word similarity between test cases and changed files (*text score*) is another mechanism to identify relevant test cases [49]. Also, test cases affected by changes in the methods they cover (*changed methods*) [37] or linked to added/removed/modified project artefacts (*project changes*) [36] between consecutive program versions can be identified and prioritised. Finally, since test cases are expected to change, their *change frequency* has also been studied to identify similar test cases [12].

3 Dimension 2: Collection

The analysis presented in this section allows identifying patterns and constraints regarding the data collection process required by each attribute.

3.1 Testing information

According to Table 1, information attributes from test cases are mostly obtained from their specification (44%), as is the case of properties like *age* or *status*, or the output after their execution (44%), which is required to build the test case *history* or determine its *effectiveness*. Only two attributes, *size* and *code similarity*, need the analysis of the test code. *Code similarity* has been computed taking execution traces as input too [38, 50, 35]. Another attribute extracted from traces is the list of *resources* used by each test case during its execution [50].

Information attributes whose value is directly observable from the source predominate in this category (value 'Simple', 67%), because most of them quantify characteristics of one test case only.Derived measurements appear when the attribute refers to dependencies or similarities that need values of every pair of test cases to be computed first. Also, the fact that the *historical effectiveness* is defined as a ratio with respect to the number of executions implies some previous calculations. Similarly, the *historical executions* attribute, which counts how many times the test case has been executed since a particular test session, should be reset every time a test case is selected again. Although some attribute values need to be periodically updated, the time and effort needed to extract and process historical data seem to be affordable compared to the cost observed in other categories [51].

Information attributes based on the test case history and report have a dynamic nature, i.e. they are session-dependent. Test case properties also changing with a certain frequency are its *status*, *execution time* and *resource utilisation*, while the *age* evolves with independence of the result of the test case. On a positive note, derived attributes analysing dependencies and similarities do not experience any change unless new test cases are incorporated to the test suite. In such a case, the pairwise values will grow incrementally, meaning that the greatest effort occurs when preparing data for the first implementation of the TCP technique.

3.2 SUT information

SUT-related information for TCP is mostly obtained by parsing the source code to an abstract representation (70%), e.g. syntax tree or dependency graphs (see Table 2). A few attributes at the component or program levels rely on the specification, whereas binary code instrumentation is required for three attributes (*change intensity*, *invocations* and input *data patterns*). From these observations, it can be concluded that the application of SUT-oriented TCP techniques is highly limited if source code is not available or it cannot be fully instrumented.

Given that classes, components and programs —the main entities for SUT-related information— can be decomposed into smaller units, several attributes are derived from the analysis of such units. However, values for the four attributes related to the program specification are directly obtained. The two change-based attributes, *buggy change* and *change intensity*, are derived but computed in very different ways. The former is predicted from 18 change metrics [20], whereas the latter compares two versions of the program to give a similarity score [21].

Class, component and program attributes are denoted as evolving, meaning that their values do not depend on whether a test case is finally executed or not, but they are subject to SUT modifications. *Fault-proneness* [52] and *buggy change* [20] attribute values can also evolve since the predictive models providing the estimation are fed with commit information. Attributes that are not expected to change (static) are those related to inputs and program specification.

3.3 Relational information

The broad range of attributes using relational information is reflected in a wider variety of sources (see Table 3). It is the only category for which all types of sources appear, and several attributes can even be obtained from more than one source. Having alternative sources makes the attribute —and therefore the corresponding TCP method— far more flexible, allowing choosing depending on how costly each source is, among those available. In this sense, coverage attributes stand out as the ones for which more sources have been explored, from specifications to traces. Compared to the previous groups, binary code and traces are reported more frequently for relational information, which suggest these attributes will require more processing. Keeping specifications of the SUT and its evolution is necessary to extract the change (*frequency* and *issues*), risk (*score* and *exposure*), and program (*test inputs*) attributes. On the other hand, fault-oriented attributes are tightly coupled to the analysis of execution results, although values for fault properties, e.g. *severity* or *age*, might come from specifications.

Since attributes establish a connection between test cases and parts or functionalities of the SUT, most of them are derived observations (89%). This is a frequent scenario for coverage-based attributes, which require identifying the elements (statements, methods, branches, etc.) exercised by each test case prior to the computation of the coverage value. The only exception is the *user-defined coverage*, whose value is directly specified by an expert [3]. Information coming from changes, faults and invocations is always derived, since these entities should be analysed to determine to what extent they affect each test case. In general, only those attributes extracted from specifications, such as *risk exposure* or *test inputs*, do not require previous measurements.

Although derived attributes prevail, most of their values will not experience changes (56%). Coverage and invocations remain static after a first analysis, unless they are defined with respect to changed methods or failing tests. Attributes related to changes and faults have an evolving or dynamic nature depending on whether their formulation is subject to the SUT lifetime (*project changes* and *fault age*) or whether it is coupled to the session (e.g. *issue score* and *fault count*), respectively. Risk estimations remain unchanged as they are derived from the specification. However, program-related attributes (*test outputs* and *usage patterns*) are expected to evolve as the SUT does.

4 Dimension 3: Representation

The third dimension characterises measurement aspects of the attributes, thus detailing how their values are computationally represented and stored.

4.1 Testing information

Attributes in Table 1 are mostly associated to numerical values (44%). This type of attribute spans across attributed related to diverse entities, such as test case properties (attributes like *age* or *size*), test case execution (*allocation* and *execution time*), and all test case similarity attributes. Nonetheless, cost and time have been expressed using categorical variables too, suggesting that the specific value is not so relevant and is rather classified into broad groups. Binary variables (22%) are a natural way to store whether the test case was executed or not, as well as its verdict (fail/pass). From these raw data, counters (integer type), ratios and weighted sums (numerical) are often derived to measure the *historical effectiveness* or the frequency of execution. Alternatives to represent test case dependencies range from binary values, indicating whether each pair of test cases should be jointly executed, to association rules describing more complex patterns.

The majority of attributes referring to test case properties are collected for each test case individually, i.e. their arity is 1:1. The exceptions are *resources*, represented as a binary list [50], and *textual description*, expressed as a bag of words or topics automatically created [49, 4]. Test case dependencies can be viewed as pairwise values (N:N) or described for one test case with respect to others (1:N). Similar to static properties, aspects surrounding test case execution are specific for each test case (1:1), and only *resource utilisation* requires multiple values. In contrast, test case history implies a sequence of values (1:N) unless only the previous session is considered. From the session outcomes it is possible to obtain the *effectiveness* of each test case (1:1). The effectiveness is used to obtain the *failure frequency* with respect to other failing tests, so the arity is 1:N. Finally, all test case similarity attributes need as many comparisons as pair of test cases exist (N:N).

4.2 SUT information

As detailed in Table 2, attributes within this group are mostly mapped to numerical (61%) and integer (26%) variables, since class and program metrics predominate. Focusing on change information, the presence of a *buggy change* is represented in binary form as it is estimated by a binary classifier [20], whereas *change intensity* is a numerical value reporting a similarity score [21]. SUT *inputs* have been studied in the context of small programs with numeric or string parameters [19, 39]. Two program-related attributes (*frequency of use* and *type of system*) are the only categorical attributes within this group, whose values are assigned by experts [18].

Focusing on the number of elements involved in the measurement, all program metrics and six class metrics (NMI, NMO, PIM, *class size*, WAC and WMC) only depend on one artefact (1:1). For the rest of class metrics (CBO, DIT, *fault-proneness*, *invocations*, LCOM, NOC and RFC), it is necessary to analyse the relation between the class and other artefacts (1:N). The same principle is applied to the single component metric (*instability*) and the prediction of *buggy changes*. *Change intensity* and *data patterns* extraction imply multiple comparisons of program versions and inputs, respectively.

4.3 Relational information

Numerical attributes are the most frequent ones for all considered entities (76%) in this group (see Table 3). One aspect to be highlighted is that the same attribute might be computed in different ways, as opposed to previous groups in which a unique formulation, and therefore type, is usually found. This happens for coverage attributes, which can be expressed as a ratio of elements covered (numerical type) or as a list of values specifying whether each element is covered or not (binary type). Similarly, *fault age* and *severity* might be represented by a number or associated to pre-established categories [4]. Definitions based on counters and frequency also appear in relational attributes, for which integer variables are appropriate.

Due to the prevalence of derived observations, the arity of relational attributes tends to be 1:N (80%) or N:N (17%). All coverage measures have arity 1:N, with the exception of *coverage distance* [31] and *historical coverage* [11] (N:N), as they compare the coverage achieved by all test cases, and *user-defined coverage* (1:1), since it is defined by an expert [3]. Fault and risk-based attributes need to associate the test case to the identified list of faults or risks [46], so they all are classified as 1:N. Attributes analysing invocations (*failing calls*) [38] and program elements (*usage patterns*) [44] are handled as N:N, since they collect values from a sequence of sessions.

Table 3: Classification of attributes based on relational information.

Source		**Collection**			**Representation**	
Entity	*Attribute*	*Artefact*	*Observation*	*Session*	*Type*	*Arity*
Change	Change frequency	SC/Spec.	Derived	Evolving	Num.	N:N
	Changed methods	BC/SC	Derived	Evolving	Int.	1:N
	Failing issues	Spec.	Derived	Dynamic	Num.	N:N
	Issue score	Spec.	Derived	Dynamic	Num.	N:N
	Project changes	BC	Simple	Evolving	Bin.	1:N
	Text score	SC	Derived	Evolving	Num.	1:N
Fault	Fault age	Exec./Spec.	Derived	Evolving	Cat./Num.	1:N
	Fault count	Exec.	Derived	Dynamic	Num.	1:N
	Fault index	BC/Spec	Derived	Dynamic	Num.	1:N
	Fault probability	BC	Derived	Static	Num.	1:N
	Fault severity	Exec./Spec.	Derived	Static	Cat./Num.	1:N
	Killed mutants	Exec.	Derived	Dyn./Sta.	Num.	1:N
Invocation	Active blocks	BC/Tra.	Derived	Static	Cat./Num.	1:N
	Change calls	BC/SC	Derived	Evolving	Int.	1:N
	Failing calls	Tra.	Derived	Dynamic	Num.	N:N
	Method calls	SC.	Derived	Static	Num.	1:N
	Relevant statements	Tra.	Derived	Static	Num.	1:N
	Sequence calls	Tra.	Derived	Static	Nom.	1:N
Program	System functions	Spec.	Simple	Static	Num.	1:N
	Test inputs	Spec.	Simple	Static	Num.	1:N
	Test outputs	Exec.	Derived	Evolving	Num.	1:N
	Usage patterns	Tra.	Derived	Evolving	Num.	N:N
Risk	Component risk	SC/Spec.	Derived	Static	Num.	1:N
	Risk coverage	BC	Derived	Static	Num.	1:N
	Risk exposure	Spec.	Simple	Static	Bin./Int.	1:N
Test case coverage	Additional coverage	BC/SC/Tra.	Derived	Dynamic	Int./Num.	1:N
	Change coverage	BC/SC/Tra.	Derived	Evolving	Bin./Num.	1:N
	Complexity coverage	BC/Spec.	Derived	Static	Num.	1:N
	Configuration coverage	Spec.	Derived	Static	Num.	1:N
	Coverage distance	BC/Spec.	Derived	Static	Num.	N:N
	Coverage frequency	BC	Derived	Dynamic	Int.	1:N
	Coverage percentage	BC/Spec.	Derived	Static	Num.	1:N
	Coverage profile	BC/Spec./Tra.	Derived	Static	Bin./Num.	1:N
	Database coverage	BC/Spec.	Derived	Static	Bin.	1:N
	Functional coverage	BC/Spec./Tra.	Derived	Static	Bin./Num.	1:N
	GUI coverage	Exec.	Derived	Static	Num.	1:N
	Historical coverage	BC/Spec.	Derived	Dynamic	Num.	N:N
	Input coverage	BC	Derived	Static	Int.	1:N
	MC/DC coverage	BC	Derived	Static	Bin.	1:N
	User-defined coverage	Spec.	Simple	Static	Cat.	1:1
	Weighted coverage	SC/Tra.	Derived	Evolving	Num.	1:N

Source: BC=Binary code, Exec=Execution results, SC=Source code, Spec=Specification, Tra=Traces
Type: Bin=Binary, Cat=Categorical, Int=Integer, Nom=Nominal, Num=Numerical

References

[1] D. Pyle, *Data Preparation for Data Mining.* USA: Morgan Kaufmann, 1999.

[2] E. Engström, P. Runeson, and A. Ljung, "Improving Regression Testing Transparency and Efficiency with History-Based Prioritization – An Industrial Case Study," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (Berlin, Germany), pp. 367–376, IEEE, 2011.

[3] S. Tahvili, W. Afzal, M. Saadatmand, M. Bohlin, D. Sundmark, and S. Larsson, "Towards Earlier Fault Detection by Value-Driven Prioritization of Test Cases Using Fuzzy TOPSIS," in *Proceedings of the 13th International Conference on Information Technology*, (Las Vegas, NV, USA), pp. 745–759, Springer, 2016.

[4] R. Lachmann, S. Schulze, M. Nieke, C. Seidl, and I. Schaefer, "System-Level Test Case Prioritization Using Machine Learning," in *Proceedings of the 15th IEEE International Conference on Machine Learning and Applications (ICMLA)*, (Anaheim, CA, USA), pp. 361–368, IEEE, 2016.

[5] S. Wang, S. Ali, T. Yue, O. Bakkeli, and M. Liaaen, "Enhancing Test Case Prioritization in an Industrial Setting with Resource Awareness and Multi-objective Search," in *Proceedings of the IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, (Austin, TX, USA), pp. 182–191, IEEE, 2016.

[6] S. Chen, Z. Chen, Z. Zhao, B. Xu, and Y. Feng, "Using semi-supervised clustering to improve regression test selection techniques," in *Proceedings of the 4th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, (Berlin, Germany), pp. 1–10, IEEE, 2011.

[7] S. Haidry and T. Miller, "Using Dependency Structures for Prioritization of Functional Test Suites," *IEEE T. Software Eng.*, vol. 39, no. 2, pp. 258–275, 2013.

[8] D. Pradhan, S. Wang, S. Ali, T. Yue, and M. Liaaen, "Employing rule mining and multi-objective search for dynamic test case prioritization," *J. Syst. Softw.*, vol. 153, pp. 86–104, 2019.

[9] K. Walcott, M. Soffa, G. Kapfhammer, and R. Roos, "TimeAware Test Suite Prioritization," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, (Portland, Maine, USA), pp. 1–12, ACM, 2006.

[10] D. Marijan, "Multi-perspective Regression Test Prioritization for Time-Constrained Environments," in *Proceedings of the IEEE International Conference on Software Quality, Reliability and Security (QRS)*, (Vancouver, BC, Canada), pp. 157–162, IEEE, 2015.

[11] J.-M. Kim and A. Porter, "A History-Based Test Prioritization Technique for Regression Testing in Resource Constrained Environments," in *Proceedings of the 24th International Conference on Software Engineering (ICSE)*, (Orlando, FL, USA), pp. 119–129, IEEE, 2002.

[12] S. Ali, Y. Hafeez, S. Hussain, and S. Yang, "Enhanced regression testing technique for agile software development and continuous integration strategies," *Software Qual. J.*, vol. 28, pp. 397–423, 2020.

[13] S. Chidamber and C. Kemerer, "A metrics suite for object oriented design," *IEEE T. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[14] H. Do, S. Mirarab, L. Tahvildari, and G. Rothermel, "The Effects of Time Constraints on Test Case Prioritization: A Series of Controlled Experiments," *IEEE T. Software Eng.*, vol. 36, no. 5, pp. 593–617, 2010.

[15] F. Touré and M. Badri, "Prioritizing Unit Testing Effort Using Software Metrics and Machine Learning Classifiers," in *Proceedings of the 30th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, (Redwood City, California, USA), pp. 653–706, KSI Research Inc, 07 2018.

[16] D. Silva, R. Rabelo, P. Neto, R. Britto, and P. Oliveira, "A Test Case Prioritization Approach Based on Software Component Metrics," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC)*, (Bari, Italy), pp. 2939–2945, IEEE, 2019.

[17] S. Panda and D. Mohapatra, "Regression test suite minimization using integer linear programming model," *Softw. Pract. Exp.*, vol. 47, no. 11, pp. 1539–1560, 2017.

[18] Q. Li and B. Boehm, "Improving Scenario Testing Process by Adding Value-Based Prioritization: An Industrial Case Study," in *Proceedings of the International Conference on Software and System Process (ICSSP)*, (San Francisco, CA, USA), pp. 78–87, ACM, 2013.

[19] F. Wang, S.-C. Yang, and Y.-L. Yang, "Regression Testing Based on Neural Networks and Program Slicing Techniques," in *Proceedings of the 6th International Conference on Intelligent Systems and Knowledge Engineering (ISKE)*, (Shanghai, China), pp. 409–418, Springer, 2012.

[20] X. Tang, S. Wang, and K. Mao, "Will This Bug-Fixing Change Break Regression Testing?," in *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, (Beijing, China), pp. 1–10, IEEE, 2015.

[21] S. Mirarab and L. Tahvildari, "A Prioritization Approach for Software Test Cases Based on Bayesian Networks," in *Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE)*, (Braga, Portugal), pp. 276–290, Springer, 2007.

[22] S. Elbaum, A. Malishvsky, and G. Rothermel, "Prioritizing test cases for regression testing," in *Proceedings of the International Symposium of Software Testing and Analysis (ISSTA)*, (Portland, Oregon, USA), pp. 102–112, ACM, 2000.

[23] G. Rothermel, R. Untch, Chengyun Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE T. Software Eng.*, vol. 27, no. 10, pp. 929–948, 2001.

[24] X. Qu, M. Cohen, and K. Woolf, “Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, (Paris, France), pp. 255–264, IEEE, 2007.

[25] H. Do, G. Rothermel, and A. Kinneer, “Empirical studies of test case prioritization in a junit testing environment,” in *Proceedings of the 15th International Symposium on Software Reliability Engineering (ISSRE)*, (Saint-Malo, Bretagne, France), pp. 113–124, IEEE, 2004.

[26] R. Rosero, O.S.Gómez, and G. Rodríguez, “Regression testing of database applications under an incremental software development setting,” *IEEE Access*, vol. 5, pp. 18419–18428, 2017.

[27] A. Nguyen, B. Le, and V. Nguyen, “Prioritizing Automated User Interface Tests Using Reinforcement Learning,” in *Proceedings of the 15th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, (Recife, Brazil), pp. 56–65, ACM, 2019.

[28] B. Miranda and A. Bertolino, “Scope-aided test prioritization, selection and minimization for software reuse,” *J. Syst. Softw.*, vol. 131, pp. 528–549, 2017.

[29] J. Jones and M. Harrold, “Test-suite reduction and prioritization for modified condition/decision coverage,” in *Proceedings IEEE International Conference on Software Maintenance (ICSM)*, (Florence, Italy), pp. 92–101, IEEE, 2001.

[30] G. Rothermel, R. Untch, Chengyun Chu, and M. Harrold, “Test case prioritization: an empirical study,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, (Oxford, UK), pp. 179–188, IEEE, 1999.

[31] B. Jiang, Z. Zhang, W. Chan, T. Tse, and T. Chen, “How well does test case prioritization integrate with statistical fault localization?,” *Inf. Softw. Technol.*, vol. 54, no. 7, pp. 739–758, 2012.

[32] S. Eghbali and L. Tahvildari, “Test Case Prioritization Using Lexicographical Ordering,” *IEEE T. Softw. Eng.*, vol. 42, no. 12, pp. 1178–1195, 2016.

[33] D. Leon and A. Podgurski, “A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases,” in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE)*, (Denver, CO, USA), p. 442, IEEE, 2003.

[34] P. Tonella, P. Avesani, and A. Susi, “Using the Case-Based Ranking Methodology for Test Case Prioritization,” in *Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM)*, (Philadelphia, PA, USA), pp. 123–133, IEEE, 2006.

[35] F. Palma, T. Abdou, A. Bener, J. Maidens, and S. Liu, “An Improvement to Test Case Failure Prediction in the Context of Test Case Prioritization,” in *Proceedings of the 14th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, (Oulu, Finland), pp. 80–89, ACM, 2018.

[36] S. Ulewicz and B. Vogel-Heuser, “Industrially applicable system regression test prioritization in production automation,” *IEEE T. Autom. Sci. Eng.*, vol. 15, no. 4, pp. 1839–1851, 2018.

[37] S. Huang, Y. Chen, J. Zhu, Z. Li, and H. Tan, “An optimized change-driven regression testing selection strategy for binary java applications,” in *Proceedings of the ACM Symposium on Applied Computing (SAC)*, (Honolulu, Hawaii), pp. 558–565, ACM, 2009.

[38] T. Noor and H. Hemmati, “Studying Test Case Failure Prediction for Test Case Prioritization,” in *Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, (Toronto, Canada), pp. 2–11, ACM, 2017.

[39] M. Roper, “Using machine learning to classify test outcomes,” in *Proceedings of the IEEE International Conference on Artificial Intelligence Testing (AITest)*, (Newark, CA, USA), pp. 99–100, IEEE, 2019.

[40] L. Zhang, J. Zhou, D. Hao, L. Zhang, and H. Mei, “Prioritizing JUnit test cases in absence of coverage information,” in *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*, (Edmonton, AB, Canada), pp. 19–28, IEEE, 2009.

[41] D. Jeffrey and N. Gupta, “Test Case Prioritization Using Relevant Slices,” in *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC)*, vol. 1, (Chicago, IL, USA), pp. 411–420, IEEE, 2006.

[42] A. Lenz, A. Pozo, and S. Vergilio, “Linking software testing results with a machine learning approach,” *Eng. Appl. Artif. Intell.*, vol. 26, no. 5, pp. 1631–1640, 2013.

[43] I. Alagoz, T. Hoiss, and R. German, “Improving System Reliability Assessment of Safety-Critical Systems using Machine Learning Optimization Techniques,” *Advances in Science, Technology and Engineering Systems Journal*, vol. 3, pp. 49–65, 01 2018.

[44] J. Anderson, M. Azizi, S. Salem, and H. Do, “On the use of usage patterns from telemetry data for test case prioritization,” *Inf. Softw. Technol.*, vol. 113, pp. 110–130, 2019.

[45] Y. Wang, Z. Zhu, B. Yang, F. Guo, and H. Yu, “Using reliability risk analysis to prioritize test cases,” *J. Syst. Softw.*, vol. 139, pp. 14–31, 2018.

[46] H. Yoon and B. Choi, “A test case prioritization based on degree of risk exposure and its empirical study,” *Int. J. Softw. Eng. Know.*, vol. 21, no. 02, pp. 191–209, 2011.

[47] L. Zhang, D. Hao, L. Zhang, G. Rothermel, and H. Mei, “Bridging the gap between the total and additional test-case prioritization strategies,” in *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, (San Francisco, CA, USA), pp. 192–201, IEEE, 2013.

[48] P. Kandil, S. Moussa, and N. Badr, “Cluster-based test cases prioritization and selection technique for agile regression testing,” *J. Softw.-Evol. Proc.*, vol. 29, no. 6, p. e1794, 2017.

[49] B. Busjaeger and T. Xie, “Learning for Test Prioritization: An Industrial Case Study,” in *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, (Seattle, WA, USA), pp. 975–980, ACM, 2016.

[50] A. Bhattacharyya and C. Amza, “PReT: A Tool for Automatic Phase-Based Regression Testing,” in *Proceedings of the IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, (Nicosia, Cyprus), pp. 284–289, IEEE, 2018.

[51] M. Khatibsyarbini, M. Isa, D. Jawawi, and R. Tumeng, “Test case prioritization approaches in regression testing: A systematic literature review,” *Inf. Softw. Technol.*, vol. 93, pp. 74–93, 2018.

[52] D. Paterson, J. Campos, R. Abreu, G. Kapfhammer, G. Fraser, and P. McMinn, “An empirical study on the use of defect prediction for test case prioritization,” in *Proceedings of the 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*, (Xian, China), pp. 346–357, IEEE, 2019.