Measuring Software Redundancy

Antonio Carzaniga*, Andrea Mattavelli* and Mauro Pezzè*†
*Università della Svizzera italiana (USI), Switzerland
†University of Milano-Bicocca, Italy
Email: antonio.carzaniga@usi.ch, andrea.mattavelli@usi.ch, mauro.pezze@usi.ch

***Abstract*—Redundancy is the presence of different elements with the same functionality. In software, redundancy is useful (and used) in many ways, for example for fault tolerance and reliability engineering, and in self-adaptive and self-checking programs. However, despite the many uses, we still do not know how to *measure* software redundancy to support a proper and effective design. If, for instance, the goal is to improve reliability, one might want to measure the redundancy of a solution to then estimate the reliability gained with that solution. Or one might compare alternative solutions to choose the one that expresses more redundancy and therefore, presumably, more reliability.**

We first formalize a notion of redundancy whereby two code fragments are considered redundant when they achieve the same functionality with different executions. On the basis of this abstract and general notion, we then develop a concrete method to obtain a meaningful quantitative measure of software redundancy. The results we obtain are very positive: we show, through an extensive experimental analysis, that it is possible to distinguish code that is only minimally different, from truly redundant code, and that it is even possible to distinguish low-level code redundancy from high-level algorithmic redundancy. We also show that the measurement is significant and useful for the designer, as it can help predict the effectiveness of techniques that exploit redundancy.

I. Introduction

A software system may implement the same functionality in different ways. For example, a container library may provide two functionally equivalent sorting algorithms: a simple in-place algorithm like insertion-sort, which is preferable for small sequences, and a more complex but asymptotically faster algorithm like merge-sort to be used on larger sequences. Even algorithmically identical operations could be performed in different ways. For example, a program may create a table of formulas in a spreadsheet by adding the first row and then copying that row vertically, or by adding the first column and then copying horizontally. A system might even contain replicas of exactly the same single operation (semantic clones) and sometimes that might be a good design choice. For example, a drawing function from a graphics library may have been refactored with only syntactic changes, but the library may also retain the old version for backward compatibility.

In all these cases—whenever a system is capable of performing the same function with different code, or even with the same code but executed in different ways—we say that the software system is redundant. Redundant elements can be present in different forms and at different granularity levels, from code snippets to entire systems. Sometimes this redundancy is introduced systematically, by design, as in the case of N-version programming, while other times it arises as a side-effect of other design goals, as in the examples above.

Software redundancy has many useful applications. In N-version programming, where the design is deliberately redundant, redundancy is intended to provide fault-tolerance. Other techniques exploit the intrinsic redundancy exemplified above, for example to avoid failures through automatic workarounds [5], to generate and deploy test oracles [4], and to overcome failures and fix faults automatically [1], [35].

However, a common limitation of all these techniques is that, while they seek to exploit redundancy, they provide no mechanism to assess how much redundancy is in fact there to be exploited. So, for example, three teams of developers may work on three independent versions of a system, but the resulting versions may turn out to be based on the same algorithm and therefore might be susceptible to correlated failures, and the developers would not be able to assess the level of independence of the three versions and therefore the level of reliability of the system.

Similarly, a self-healing system may try to deploy an automatic workaround for a failing method by replacing that method with a supposedly equivalent but hopefully non-failing alternative. But the fundamental question remains: how much redundancy is there in the chosen alternative? And therefore, how likely is it that the chosen alternative would avoid the failure? More specifically, is the alternative executing different code or is it merely a thin wrapper for the same code? Even if the code is different, is the alternative radically different, or is it using essentially the same algorithm? More fundamentally, how do we even quantify redundancy?

With this paper, we intend to provide answers to these questions. We begin by formulating a very general and abstract notion of redundancy, which in essence we define as functional equivalence with execution diversity. Then, again abstractly, we define a measure of redundancy in terms of the dissimilarity between the execution traces of equivalent code fragments, computed over a certain type of execution traces, and aggregated over a space of executions.

These abstract notions of redundancy and its measure are conceptually important but they are also ultimately undecidable and therefore not directly usable in practice. Notice in fact that equivalence alone is undecidable, and that the dissimilarity measure should be computed over an infinite space of executions. We therefore develop a method to measure redundancy that is based on the abstract model but is also practical and in fact quite efficient. The method considers a finite set of execution

traces obtained from generated input values (generated tests), uses a specific projection of the execution trace in which we log changes in the application state (memory accesses), and measures a specific form of edit distance between traces.

We first validate our method through micro-benchmarks to show that the measurement is internally consistent and stable with respect to some forms of program transformations. We then use the method to take measurements in a number of case studies to assess the significance of the measure. The results show that the measurement is a very good indicator of redundancy. In particular, the measurement clearly distinguishes shallow differences where two apparently different code fragments reduce to the same code underneath, from deep code differences, from algorithmic differences where not only the code is different but the algorithmic nature of the two executions differs significantly. We also demonstrate that our measures of redundancy can be good predictors of the effectiveness of techniques that exploit redundancy, such as automatic workarounds.

In summary this paper contributes a complete and consistent definition of redundancy for software systems, and develops and evaluates a practical method to measure redundancy quantitatively. We see this as a first foundational step in a broader study of software redundancy. We see redundancy as a precious resource that can and should be exploited to improve software quality. In particular, we believe that many software engineering techniques can produce an added value only to the extent that they can find and exploit information within the software itself. In other words, those techniques feed on the redundancy of software. It is therefore important to study redundancy, at a minimum to characterize it qualitatively and quantitatively in specific cases, but then also to generalize its characterization, perhaps to identify properties of a design or development process that lead to more or less redundancy.

II. Background and Motivation

Redundancy has many applications in software engineering, including fault tolerance [2], [30], self-healing systems [5], [6], self-adaptive services [10], [14], [22], [24], [31], [33], [34], self-checking programs [36], self-optimizing code [9], [25], automatic fault fixing [1], [35], and the automatic generation of test cases and oracles [4]. It is exploited at different abstraction levels, from a service or subsystem [14] to a method call [5], [11], [16]. Some techniques and applications deliberately introduce redundancy within a system, as in the case of N-version programming and recovery blocks [2], [30], while other techniques exploit redundancy available outside the system, for example in a marketplace of Web services [10], [33], [34], and yet other techniques seek to exploit the redundancy that is intrinsic in software systems [5], [6].

We now briefly review a few relevant applications of redundancy in the domains of software reliability, self-healing systems, and automated fault fixing. We do that to motivate the idea of measuring redundancy. Then we review the research work most related to that idea.

N-version programming is a well-known technique to meet the reliability requirements of safety critical applications [2]. An N-version system consists of multiple independently developed implementations (of the same system) that execute in parallel on the same input. The system then outputs the result of the majority of the versions, so the gain in reliability rests on the assumption that coincidental faults in independently developed components are very unlikely. The validity of this assumption has been disputed, most notably by Knight and Leveson [21] who argue that independently developed components may not be as independent—or as *redundant*—as one would expect. This suggests that a consistent measure of redundancy would support the engineering of N-version systems by at least identifying weak components.

Automatic workarounds exploit redundancy in methods and code fragments to achieve a form of self-healing [5]. The idea is to identify redundant code fragments that can replace failing fragments at runtime to automatically circumvent a fault and avoid the failure. The technique relies on the amount of redundancy of alternative code fragments. A measurement of that amount could indicate the likelihood that a particular alternative fragment would succeed in avoiding the failure, thus leading to a more efficient self-healing mechanism.

A recent approach to fixing faults automatically is to use genetic programming, to "evolve" a faulty code towards a correct one [35]. Thus redundancy plays a crucial role also in this case, because some of the mutations through which the system evolves replace a fragment of code with another fragment taken from the system itself. Therefore, a measure of redundancy could indicate the likelihood of success for the entire evolutionary process, and it could also be used as a fitness function for the selection of an individual mutation.

Measuring software redundancy seems like a useful proposition, but it also remains an open problem. Some researchers have proposed ideas and concrete techniques to assess the semantic similarity of code fragments within a system, which is an essential ingredient of redundancy. In particular, the most relevant work is that of Jiang and Su, who consider the problem of identifying semantic clones [20]. Their definition of clones is based on a notion of equivalent code fragments similar to the one we also develop in this paper. However, it is in the definition of equivalence relation that our model differs significantly. In essence, Jiang and Su consider fragments with identical effects, whereas we consider two fragments to be equivalent when their *observable* effects are identical. We re-examine this difference in more detail in Section III-A.

Another related work is described in a recent paper by Higo and Kusumoto on the "functional sameness" of methods [18]. Here too the authors are interested in the semantic similarities between methods within a system. However, their approach differs almost completely from ours (and also from Jiang and Su), since they use an exclusively *static* analysis of the code, and furthermore they do that by combining three measures of similarity that, in and of themselves, have little to do with the actual semantics *of the code.* Specifically, they use the similarity of the vocabulary of variables (symbols), the name

of the methods, and the "structural" distance of methods within the package hierarchy. The most interesting aspects of the work of Higo and Kusumoto is that, perhaps thanks to the simplicity of their analysis, they were able to analyze a massive amount of code.

III. Characterizing Software Redundancy

We now formalize our notion of redundancy. We first give an abstract and general definition that we then specialize to develop a practical measurement method.

We are interested in the redundancy of code. More specifically we define redundancy as a relation between two code *fragments* within a larger system. A code fragment is any portion of code together with the necessary linkage between that code and the rest of the system. A fragment can be seen as the in-line expansion of a function with parameters passed by reference, where the parameters are the linkage between the fragment and its context. Figure 1 illustrates the notion of redundant fragments with two examples. For each pair of fragments, we specify the linkage between the fragments and the rest of the system by listing the variables in the fragments that refer to variables in the rest of the system. All other variables are local to the fragments.

linkage: int x; int y;	

int tmp = x;
x = y;
y = tmp;
	x ^= y;
	y ^= x;
	x ^= y;

linkage: AbstractMultimap map; String key, Object value;	

map.put(key, value);	List list = new ArrayList();
	list.add(value);
	map.putAll(key, list);

Fig. 1. Examples of redundant code fragments.

The first example (first row) shows a code fragment that swaps two integer variables using a temporary variable (left side) and another fragment that swaps the same variables without the temporary by using the bitwise xor operator (right side). The second example refers to a multi-value map in which one can add an individual mapping for a given key (*put*) or multiple mappings for the same key (*putAll*). In both cases, the fragment on the left is different from, but equivalent to, the fragment on the right. This is our intuitive definition of redundancy: two fragments are redundant when they are *functionally equivalent* and at the same time their *executions are different.* We now formalize these two constituent notions.

A. An Abstract Notion of Redundancy

We want to express the notion that one should be able to replace a fragment A with a redundant fragment B, within a larger system, without changing the functionality of the system. This means that the execution of B would produce the same results as A and would not cause any noticeable difference in the future behavior of the system. In other words, we want B to have the same result and equivalent side-effects (or state changes) as A.

Other studies on semantically equivalent code adopt a purely functional notion of equivalence, and therefore assume no visible state changes [11]. Yet others consider state changes to be part of the input/output transformation of code fragments, but then accept only identical state changes [20]. Instead, we would still consider two fragments to be equivalent even if they produce *different* state changes, as long as the *observable effects* of those changes are identical. This notion is close to the testing equivalence proposed by De Nicola and Hennessy [27] and the weak bi-similarity by Hennessy and Milner [17]. We now formulate an initial definition of equivalence between code fragments similar to testing equivalence.

1) Basic Definitions: We model a system as a state machine, and we denote with $\mathbb{S}$ the set of states, and with $\mathbb{A}$ the set of all possible actions of the system. The *execution* of a code fragment C starting from an initial state S_0 amounts to a sequence of actions $\alpha_1, \alpha_2, \ldots, \alpha_k \in \mathbb{A}$ that induces a sequence of state transitions $S_0 \xrightarrow{\alpha_1} S_1 \xrightarrow{\alpha_2} \cdots \xrightarrow{\alpha_k} S_k$. In this model we only consider code fragments with sequential and terminating and therefore finite (but unbounded) executions, and without loss of generality we consider the input as being part of the initial state.

We then use $\mathbb{O}$ to denote the set of all possible outputs, that is, the set of all externally observable effects of an execution. We use $Out(S, \alpha) \in \mathbb{O}$ to denote the output corresponding to the execution of action α starting from state S, and, generalizing, we denote with $Out(S_0, C) \in \mathbb{O}^*$ the output of the sequence of actions $\alpha_1, \alpha_2, \ldots, \alpha_k$ corresponding to the execution of C from state S_0.

2) Observational Equivalence: We say that two code fragments C_A and C_B are *observationally equivalent* from an initial state S_0 if and only if, for every code fragment C_P (probing code), the output $Out(S_0, C_A; C_P)$ is the same as $Out(S_0, C_B; C_P)$, where $C_A; C_P$ and $C_B; C_P$ are code fragments obtained by concatenating C_A and C_B with the probing code C_P, respectively.

This definition requires that the two code fragments and the follow-up probing code produce exactly the same output, which does not take into account the intended semantics of the system whereby different output sequences may be equally valid and therefore should be considered equivalent. For example, consider a container that implements an unordered set of numbers that in state S_0 represents the set $\{10\}$. Consider now a fragment C_A that adds element 20 to the set, and a supposedly equivalent fragment C_B that also adds 20 to the set but with a different internal state transformation: C_A leaves the set in a state such that an iteration would first go through 10 and then 20, while C_B causes the same iteration to first go through 20 and then 10. C_A and C_B would not be considered observationally equivalent according to the definition above, since a probing code that iterates through the elements of the set would expose a difference.

To account for the semantics of the system, we consider a more general definition that requires the output of the two

fragments and the follow-up probing code to be equivalent according to an oracle relation that embodies the semantics of the system. Let $T_{S_0,C_A,\cdot}$ be a family of equivalence relations (hereafter *oracles*) that, depending on S_0 and C_A, and for every valid probing code C_P defines an equivalence oracle $T_{S_0,C_A,C_P} \subseteq \mathbb{O}^* \times \mathbb{O}^*$ that essentially says whether two output sequences are both correct (and therefore equivalent) for the execution of $C_A;C_P$ from S_0. Thus we say that C_A and C_B are observationally equivalent in a semantically meaningful sense, from an initial state S_0, when, for every probing code fragment C_P, the output sequences $Out(S_0,C_A;C_P)$ and $Out(S_0,C_B;C_P)$ are equivalent according to the oracle. More specifically, we say that C_B *can replace* C_A when the outputs are equivalent according to T_{S_0,C_A,C_P}, and vice-versa C_A can replace C_B when the outputs are equivalent according to T_{S_0,C_B,C_P}. Finally, we say that C_A and C_B are equivalent when they can replace each other. Notice that the former and more strict definition is a special case in which the oracles reduce to the identity relation.

3) Redundancy: Putting the pieces together, we say that two code fragments C_A and C_B are redundant in state S_0 when they are observationally equivalent from state S_0 in a semantically meaningful sense, and their executions from S_0 differ, meaning that the sequence of actions $\alpha_{A,1}, \alpha_{A,2} \ldots$ induced by C_A (from S_0) differs from the sequence $\alpha_{B,1}, \alpha_{B,2} \ldots$ induced by C_B (from S_0). We then say that C_A and C_B are always redundant if they are redundant in every state S_0 in which their application is valid (every syntactically valid application of the fragments with every valid input).

B. Discussion and Practical Considerations

This model of redundancy is built upon our practical experience in developing various concrete techniques to capture and exploit software redundancy [4], [5], [6], [15], and generalizes to other uses of software redundancy such as N-version programming [3] and recovery blocks [30]. However, our main objective now is to abstract from each technique and to capture the essence of software redundancy, and this is why we formulated an abstract and general model. We see this as a first necessary step towards understanding the nature of redundancy in software systems and to use it systematically to improve the reliability of software systems.

From this abstract model, we then want to derive a concrete method to characterize the redundancy of a system. In particular, we would like to obtain a measurement that would somehow correlate with the attainable benefits of the redundancy present within a system. To do that, we need to overcome two obstacles. First, our abstract notion of redundancy is not decidable, since it subsumes a basic form of equivalence between programs that amounts to a well-known undecidable problem (by Rice's theorem). We therefore need to either limit the expressiveness of the model or somehow accept an incomplete or imprecise decision procedure. Second, our model expresses a binary decision, but we would like a more informative measure, for example to rank different designs by the *amount* of redundancy they possess. We must therefore enrich the model with a form of distance and we must define a corresponding operational measurement method.

The first problem (deciding equivalence) has been studied extensively from a theoretical perspective independent of its relation to the notion of redundancy. We also explored the same problem from a practical perspective and specifically in relation to redundancy. In particular, we initially proposed a completely manual method to simply express equivalence in the form of rewriting rules that we then used to annotate potential sources of redundancy in code bases of significant size and complexity [5], [6]. We then developed a method to automatically test the equivalence of code fragments in the specific context of test oracles using a bounded search. In this paper we enhance this bounded-search method to decide equivalence and therefore redundancy.

The second problem (obtaining a non-binary measurement of redundancy) is our main focus here. We discuss it next.

C. A Practical and Meaningful Measure of Redundancy

Recall once again that redundancy is present whenever two code fragments induce different executions with the same functional effect on a system. We now want to extend this abstract binary condition, which is defined for each starting state S_0, to obtain a more general and meaningful measurement of the redundancy of two code fragments. By "meaningful" we mean a measurement that can serve as a good indicator or predictor for software designers. In particular, we consider a measure of redundancy to be meaningful when it is indicative of some useful property or some design objective related to redundancy. For example, if we use redundant fragments in an N-version programming scheme to increase the reliability of a system, then a meaningful measurement of their redundancy should correlate with the gain in reliability attainable with that scheme. Here we formulate a general method that can be specialized with several different metrics, and later in Section IV we experimentally evaluate the predictive ability of each specialization.

We define a meaningful measure of redundancy by first turning the binary condition into a richer non-binary metric, and then by aggregating this metric over several starting states. Informally, given two code fragments, we combine a measure of the *degree of functional equivalence* of the fragments with a measure of the *distance* between their executions, and then aggregate the results over a set of representative initial states.

A bit more formally, let $e_S(C_A,C_B) \in [0,1]$ denote the degree of equivalence between two code fragments C_A and C_B in state S. Intuitively, $e_S(C_A,C_B)$ can be seen as the probability that a probing code C_P would not expose any difference between C_A and C_B when starting from state S. Also let $d_S(C_A,C_B) \in [0,1]$ denote the distance (normalized) between the executions of C_A and C_B starting from S. Thus $d_S(C_A,C_B) = 0$ indicates two identical executions while $d_S(C_A,C_B) = 1$ indicates completely different executions.

With these equivalence and distance measures, we define the (generic) measure of the redundancy of C_A and C_B, in state S, as $R_S = e_S(C_A,C_B)d_S(C_A,C_B)$. Then, ideally, we would like to

compute the expected value of R_S over the space of all possible initial states S. In practice, we aggregate over a limited sample of the state space.

input: code fragments C_A, C_B		
	repeat n times:	
1:	$S \leftarrow$ *choose from* $\mathbb{S}$	// sample the state space
2:	$e \leftarrow e_S(C_A, C_B)$	// measure equivalence
3:	$d \leftarrow d_S(C_A, C_B)$	// measure difference
	$R_S \leftarrow e \cdot d$	
4:	**return** *aggregate all* R_S	// R_S in expectation

Fig. 2. General algorithm to measure redundancy.

In summary, we measure the redundancy of two fragments using the algorithm of Figure 2. We now present the techniques that implement steps 1, 2, 3, and 4 of the algorithm.

1) Sampling of the State Space: We must choose and set up a number of initial system states from which to execute the two code fragments C_A and C_B. This amounts to generating tests for the system in which, at some point, we can insert the two fragments (either one). To do that, we employ two common testing methods, namely random testing and input category partitioning.

The main method we use is random test input generation. In principle, once we have a test, we could insert one of the two fragments in almost any position in the test. However, this would require some care in linking the fragments with the system. So, to automate this process completely, we let the test generator itself insert the fragments directly into the test. In practice, we generate many tests and select those that already contain C_A, and for those with multiple instances of C_A we consider as an initial state the state right before each instance of C_A.

In addition to random testing, in some special cases, we also manually categorized the input to the system so as to represent classes of equivalent inputs. We then compile a list of tests that cover each category. We then use these tests as operational definitions of the initial state.

2) Observational Equivalence Measure: We compute the degree of observational equivalence $e_S(C_A, C_B)$ by directly applying its definition: we generate a large number of probing code fragments C_P, which we execute right after C_A and C_B, respectively, from state S, and we compare the output for each pair of executions. We then return the percentage of cases in which the outputs are the same.

In essence, probing code fragments amount, once again, to random tests, specifically for the variables corresponding to the linkage of the two fragments. We therefore implement a specialized random test generator that starts from a pool of variable descriptors, each indicating name and type. At each step, the generator selects a variable from the pool together with a public method to call on that variable, and adds that call to the test. Then, if the method returns a primitive value, or if the variable is itself of a primitive type, then the generator adds a statement to output the value. Otherwise, if the method returns another object, the generator assigns the result of the call to a newly declared variable, and also adds a descriptor for the new variable (name and type) to its pool of variables. The generator also adds the necessary code to catch and output any exception that might be raised at each step of the test. The generator terminates and outputs the generated test after a preset number of steps. Figure 3 shows an example of a generated probing code fragment for a code fragment similar to the second example of Figure 1.

```
// ... testing code to set up initial state...

// Code fragment A (linkage: boolean result; ArrayListMultimap map;)
boolean result = map.put(var1, var2);

// generated probing code:
System.out.println(result);
boolean x0 = map.isEmpty();
System.out.println(x0);
map.clear();
java.util.Map x1 = map.asMap(); // x1 added to the pool
int x2 = map.size();
System.out.println(x2);
int x3 = x1.size();
System.out.println(x3);
java.util.Set x4 = x1.entrySet(); // x4 added to the pool
java.util.Iterator x5 = x4.iterator(); // x5 added to the pool
boolean x6 = x4.isEmpty();
System.out.println(x6);
try {
  x5.remove();
} catch (java.lang.IllegalStateException e) {
  System.out.println(e);
}
```

Fig. 3. Example of a generated probing code executed immediately after one of the code fragments (A) under measurement. (The code is simplified and abbreviated for presentation purposes.)

Notice that we compare the output of the generated probing code fragments using a simple equality test that in general leads to a conservative form of observational equivalence (as described in Section III-A2). Still, this method is both practical and efficient, and it is also exact (no false negatives) for all the subjects considered in our evaluation (Section IV).

3) Difference Between Executions: We define a distance measure $d_S(C_A, C_B)$ by applying a dissimilarity measure to a projection of the executions of the two code fragments C_A and C_B starting from state S. We define and experiment with many such distance measures by combining various dissimilarity measures with various projections. A projection of an execution $\alpha_1, \alpha_2, \ldots, \alpha_k$ is a particular trace in which we log a subset of the information associated with each action α_i. In particular, we use two categories of projections.

In *code projections*, for each action we log the code executed in that action. We experimented with a number of code projections in which we log a simple identifier of the line of source code, the line of code plus its depth in the call stack, or to the full call stack.

In *data projections*, for each action we log the read/write operations performed in that action. We only log read or write operations on object fields or static fields. Both read and write entries, logged individually, consist of an address part that identifies the field from which we read or into which we write, and a data part that identifies the value being read or written. We then specialize this projection by encoding different pieces

of information in the address and data parts. For the address we use the field name, the class name, or the type of the field being read or written, and a number of their combinations. For the value we use its string representation for basic types or arrays of basic types, or no value at all.

We experimented with various combinations of code and data projections, and also with slightly more elaborate variants of each. For example, for write operations we log the old value as well as the new value. Table I summarizes the most significant projections we propose to use. The examples refer to the simple code fragment `boolean x = map.put(k,v)`. We evaluate the effectiveness of these projections in Section IV.

TABLE I
PROJECTIONS USED TO DERIVE AN ACTION LOG FROM AN EXECUTION

Type	Projection	Example from an actual log[(a)]
Code	statement	ArrayListMultimap.put(LObject;LObject;)Z@66
		AbstractListMultimap.put(LObject;LObject;)Z@95
		AbstractMultimap.put(LObject;LObject;)Z@200
	depth, statement	3:ArrayListMultimap.put(LObject;LObject;)Z@66
		4:AbstractListMultimap.put(LObject;LObject;)Z@95
		5:AbstractMultimap.put(LObject;LObject;)Z@200
Data	type, value	Ljava/util/Map;→{}
		Ljava/util/Set;→[]
		Ljava/util/HashMap;→{}
		I→1
		I←1
	field, value	map→{}
		map→{}
		entrySet→[]
		this$0→{}
		modCount→1
		expectedModCount←1
	class, field, value	AbstractMultimap.map→{}
		HashMap.entrySet→[]
		HashMap$EntrySet.this$0→{}
		HashMap$HashIterator.modCount→1
		HashMap$HashIterator.expectedModCount←1
	type, old value	map→{}
		entrySet→[]
		this$0→{}
		modCount→1
		expectedModCount 0←1
	no value	AbstractMultimap.map→
		HashMap.entrySet→
		HashMap$EntrySet.this$0→
		HashMap$HashIterator.modCount→
		HashMap$HashIterator.expectedModCount←

[(a)] code fragment: `boolean x = map.put(k,v);`
abbreviations:
ArrayListMultimap is com.google.common.collect.ArrayListMultimap
AbstractMultimap is com.google.common.collect.AbstractMultimap
HashMap is java.util.HashMap
Object is java.lang.Object

To obtain a clean execution log, we also discard log entries corresponding to platform-related actions that we consider irrelevant and potentially confusing for our purposes. For example, in our implementation, which is in Java, we log the actions of the Java library but we discard those of the class loader. With such logs we then proceed to compute the difference measure for code fragments.

Let $L_{S,A}$ and $L_{S,B}$ be the logs of the execution of fragments C_A and C_B from state S. We generally compute the distance measure $d_S(C_A, C_B) = 1 - similarity(L_{S,A}, L_{S,B})$ where *similarity* is a normalized similarity measure defined over sequences or sets (interpreting the logs as sets). Intuitively, the normalization of the similarity measures takes into account the length of the logs, but in general each measure has its own specific normalization procedure. Notice that in the application of the similarity measure, we consider each entry as an atomic value that we simply compare (equals) with other entries. Table II lists the most effective similarity measures we experimented with. (The abbreviations on the right side identify the measures in their experimental evaluation in Section IV.)

TABLE II
SIMILARITY MEASURES APPLIED TO ACTION LOGS

sequence-based	Levenshtein [26]	(Lev)
	Damerau–Levenshtein [26]	(DamLev)
	Needleman–Wunsch [32]	(Need)
	Cosine similarity [8]	(Cos)
	Jaro [19]	(Jaro)
	Jaro–Winkler [19]	(JaroW)
	Q-grams [8]	(qGrams)
	Smith–Waterman [12]	(SmithW)
	Smith–Waterman–Gotoh [12]	(SmithG)
	Overlap coefficient [7]	(Ovlp)
set-based	Jaccard [8]	(Jaccard)
	Dice [8]	(Dice)
	Anti-Dice [8]	(ADice)
	Euclidean [7]	(Euclid)
	Manhattan [7]	(Man)
	Matching coefficient [7]	(MC)

4) Aggregating the Measure of Redundancy: Ideally we would like to obtain the *expected* redundancy of two fragments C_A and C_B. However, we also want to use a general aggregation method, independent of the particular distribution of the input for the system at hand. We therefore simply compute the average of the redundancy measures over the sample of initial states. We also experimented with other obvious aggregation functions, such as minimum, maximum, and other quantiles, but those proved less effective than the simple average.

IV. EXPERIMENTAL VALIDATION

We conduct an experimental analysis to validate the measurement method we propose. We consider two validation questions. The first question (Q1) is about consistency: we want to check that our measurements are internally consistent. In particular, we want to verify that the method yields measurements that remain stable when using common semantic-preserving code transformations (such as refactoring) and also when we sample the state space from a domain of semantically similar inputs.

The second question (Q2) is about significance: we want to make sure that the measurements we obtain are meaningful and useful. In general we want to show that those measurements can help developers make design decisions related to the redundancy of their system. Thus we judge their significance and their utility by correlating the measurements with specific uses of redundancy.

A. Experimental Setup

We conduct a series of experiments with a prototype implementation of the measurement method described in

Section III-C. We consider a number of subject systems (described below) for which we consider a number of pairs of code fragments (also detailed below). For each system we generate a set of test cases either manually using the category partition method [28] or automatically with either Randoop [29] or Evosuite [13] depending on the experiment. We use the tests generated for each system with all the pairs of code fragments defined for that system to sample the input space as described in Section III-C1. Then for each pair of fragments and for each initial state (test) we measure the degree of observational equivalence as explained in Section III-C2. For each pair of fragments and initial state, we also trace the executions of the two fragments using the DiSL instrumentation framework [23]. With these traces, we then apply a number of projections as explained in Section III-C3, and with the resulting logs we compute a number of similarity measures using a modified version of the SimMetrics library.[1] Finally, we compute the redundancy measure for each initial state, and aggregate with its overall average and standard deviation.

We experiment with two sets of programs:

- *Benchmark 1* is a set of different implementations of two search algorithms and four sorting algorithms. Table III lists each algorithm and the number of corresponding implementations. In the case of Benchmark 1 we consider each whole system as a code fragment and we compare code fragments of the same category. For example, we may compare one implementation of bubble sort with one of quicksort, or two implementations of binary search.
- *Benchmark 2* is a set of classes of the Google Guava library. The set of Guava classes contains methods that can be substituted with other equivalent code fragments (equivalent according to the documentation). We therefore consider all pairs of fragments consisting of a method (C_A) and an equivalent fragment (C_B). Table IV lists all the subject class with the number of methods for which we have equivalent fragments, and the total number of equivalent fragments.

TABLE III
BENCHMARK 1: DIFFERENT IMPLEMENTATIONS OF SEARCH AND SORTING ALGORITHMS

Algorithm	Implementations
Binary search	4
Linear search	4
Bubble sort	7
Insertion sort	3
Merge sort	4
Quicksort	3

B. Internal Consistency (Q1)

We check the internal consistency of the measurement method with three experiments. The first experiment checks the obvious condition that a fragment is not redundant with itself (since the two executions should be identical). We conduct this experiment using the code fragments (programs) of Benchmark 1, for which we indeed obtain a redundancy measure of 0.

[1] http://sourceforge.net/projects/simmetrics/, the modifications are to reduce the space complexity of several measures. For example, the Levenshtein distance in SimMetrics uses $O(n^2)$ space, which is necessary to output the edit actions that define the edit distance. However, since we only need to compute the numeric distance, we use a simpler algorithm.

TABLE IV
BENCHMARK 2: GUAVA CLASSES, SOME OF THEIR METHODS AND THEIR EQUIVALENT IMPLEMENTATIONS

Library	Class	Methods	Equivalences
Guava	ArrayListMultimap	15	23
	ConcurrentHashMultiset	17	29
	HashBimap	10	12
	HashMultimap	15	23
	HashMultiset	17	29
	ImmutableBimap	11	17
	ImmutableListMultimap	11	16
	ImmutableMultiset	9	27
	Iterators	1	2
	LinkedHashMultimap	15	23
	LinkedHashMultiset	18	30
	LinkedListMultimap	15	23
	Lists	8	20
	Maps	12	29
	TreeMultimap	13	21
	TreeMultiset	17	29

With the second experiment we check that the measurement is stable with respect to semantic-preserving program transformations in code fragments, as well as with semantically irrelevant changes in the input states. We use once again the fragments of Benchmark 1. In a first set of experiments, we first apply all the automatic refactoring operations available within the Eclipse IDE (Extract to Local Variable, Extract Method, Inline Expression and Change Name) as many times as they are applicable, but only once on each expression. We then measure the redundancy between each original fragment and its refactored variants.

Figure 4 shows a minimal but indicative subset of the results of the refactoring experiments (all other results are consistent). We plot the redundancy measure for the binary search case study, with data and code projections in histograms corresponding to all the refactoring operations, identified by the color of the bar. The X-axis indicates the similarity measure used to compute the redundancy (see Table II for the abbreviations of the similarity measures).

We notice immediately that code projections are inconsistent, and are negatively affected by essentially all refactoring operations under every similarity measure. By contrast, data projections have an excellent consistency and stability, and correctly report zero or near-zero redundancy under all refactorings and with all similarity measures. An analysis of the results reveals that data projections based on type rather than field name are particularly robust for some refactoring activities, such as Extract-method and Change-name, and less robust with respect to others that may change the execution actions. For example, if we apply the Extract-local-variable operator to the variable in a for loop condition that checks the length of an array field, then that changes the number of field accesses and thus the data projections.

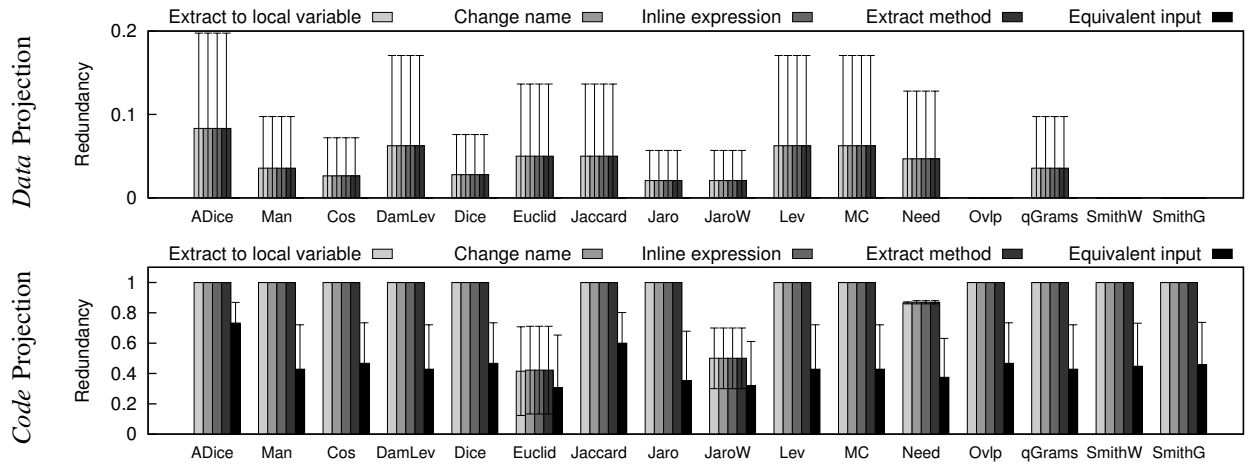

Fig. 4. Consistency of the redundancy measure using various projections on binary search. Notice that the scale of the first chart is from 0 to 0.2. This is to better highlight the fact that the results for the *data* projection are very low, either exactly zero or very close to zero.

In a second consistency check, we consider the redundancy measured between executions of the same fragment, but starting from a different but semantically equivalent state (test case). This experiment does not conform to our definition of redundancy. Still, it can be useful to test the stability of the measurement with respect to small (insignificant) variations in the initial state. For these experiments we use the test cases that select the initial states within the same input category (input category partitioning). We report the results of these experiments also in Figure 4 in the rightmost bar in each group (darkest color, labeled "Equivalent input"). The results are in line with those of the other consistency checks, that is, data projections have excellent consistency and stability, while code projections are inconsistent.

TABLE V
OBSERVATIONAL EQUIVALENCE FOR THE METHODS OF THE GUAVA CLASS ARRAYLISTMULTIMAP

Method	Equivalence (average)
clear()	1.00
containsEntry(Object,Object)	1.00
containsKey(Object)	1.00
containsValue(Object)	1.00
create()	1.00
create(int,int)	1.00
isEmpty()	1.00
keys()	0.61 (median: 0.57)[a]
put(Object,Object)	1.00
putAll(Multimap)	1.00
putAll(Object,Iterable)	1.00
remove(Object,Object)	1.00
removeAll(Object)	1.00
replaceValues(Object,Iterable)	1.00
size()	1.00

[a]see detailed measurements in Table VI

With the third consistency experiment, we focus specifically on the measure of the degree of equivalence, which corresponds to the probability that a probing code would not reveal a difference (see Section III-C2). For this experiment we can not use the fragments from Benchmark 1, since we know that those are all equivalent. We therefore focus on the particular case of the *ArrayListMultimap* class taken from Benchmark 2. Table V lists all the methods that define our first code fragment (C_A) for *ArrayListMultimap*. For each one of them, we then measure and report in the table the degree of observational equivalence with all the corresponding fragments from Benchmark 2 (average over several other fragments C_B and over all initial states). The degree of equivalence is exactly 1 for all methods, which is what we expected, except for method *keys*, which is paired with a fragment that uses *keySet*.

TABLE VI
EQUIVALENCE MEASUREMENT OF METHODS *keys()* AND *keySet()*

Initial state	generated C_P	failed C_P	Equivalence
S_1	13	6	0.54
S_2	16	5	0.69
S_3	16	5	0.69
S_4	15	5	0.67
S_5	14	6	0.57
S_6	16	7	0.56
S_7	16	7	0.56
Average:	15.14	5.86	0.61

A closer examination indicates that *keys* and *keySet* are similar but not completely equivalent, since they differ when the multimap contains multiple values for the same key. To better analyze the case, we repeated the experiment with 106 probing codes C_P starting from 7 different initial states $S_1, \ldots, S_7$. The results show that the measurement correctly quantifies the differences in the sequences of actions, and that the results are consistent across initial states and probing codes.

In summary, the results of the three experiments described so far demonstrate the internal consistency and robustness of the measurement. These experiments were also essential to identify the best projections and similarity measures. In the following experiments, we use only *data* projections with a few of the most effective similarity measures. When not indicated, the similarity measure is the Levenshtein distance.

C. Significance (Q2)

We evaluate the significance of the redundancy measurements obtained through our method in two ways. We first assess the ability of the measurement to identify differences (redundancy) at various levels of abstractions, and more specifically low-level code redundancy versus high-level algorithmic redundancy. We conduct this series of experiments on the cases of Benchmark 1. Then, in a second series of experiments based on some cases taken from Benchmark 2, we assess the ability of the redundancy measure to predict the effectiveness of a particular technique designed to take advantage of redundancy.

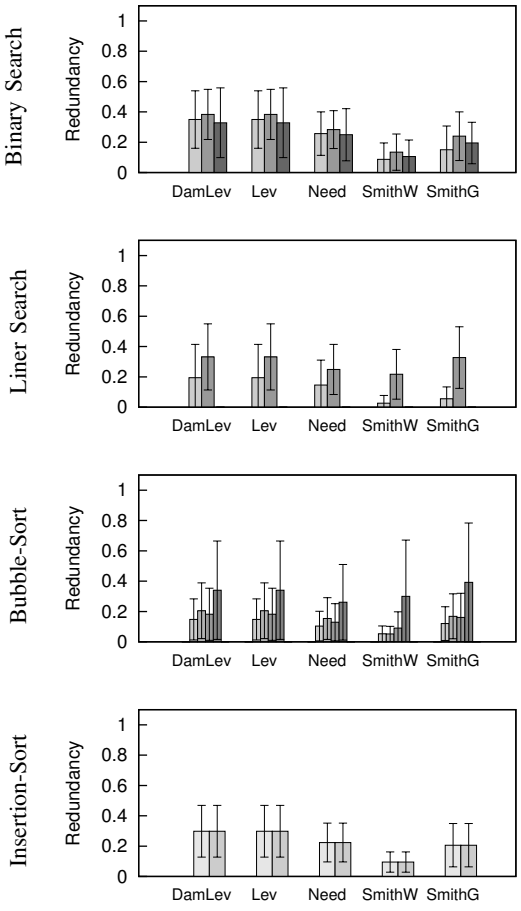

Fig. 5. Redundancy between implementations of the same algorithm.

Figure 5 shows the measurements of the redundancy between fragments that implement exactly the same algorithm. Each plot shows groups of values representing the average and standard deviation over the measurements between each implementation and every other implementation. However, notice that in some cases the results are always zero, and therefore do not appear in the histogram. So, for example, the first histogram shows the case of the four implementations of binary search (see Table III), one of which has zero redundancy, and therefore the histogram shows three bars for each similarity measure. In general, the redundancy measures are low, which makes sense, since all the fragment pairs implement the same algorithm and can only have low-level code differences.

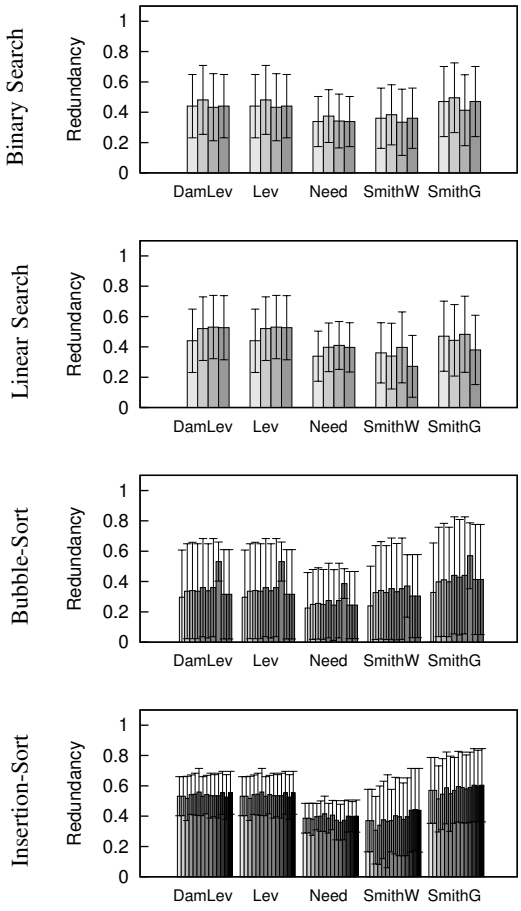

Fig. 6. Redundancy between different algorithms.

Figure 6 shows the redundancy between fragments that implement different algorithms. Here each histogram represents one particular algorithm, and shows the comparisons between each implementation of that algorithm and every other implementation of another algorithm in the same category (sorting and searching). Here the measures are relatively high, indicating a high degree of redundancy, which makes sense, since all the fragment pairs implement different algorithms, and therefore should have significant differences.

The last series of results we present are intended to assess the significance of the measurement in terms of its predictive ability. For this we analyze the redundancy of some equivalent fragments that we used as automatic workarounds with a technique intended to increase the reliability of systems [5]. In particular, we consider a number of pairs of equivalent code fragments used with varying degrees of success as workarounds. We then measure the redundancy for each pair, and see how that correlates with the success ratio.

Table VII shows the results of these experiments. For each subject system we sort the equivalent pairs by their success ratio to highlight the correlation with the measure of redundancy.

TABLE VII
CORRELATION BETWEEN REDUNDANCY MEASURE AND THE EFFECTIVENESS OF AUTOMATIC WORKAROUNDS

System	Method (C_A)	Workaround (C_B)	Success ratio	Redundancy
Caliper	Iterators.forArray(a)	Arrays.asList(a).iterator()	3/3 (100%)	1.00 ± 0.00
	LinkedHashMultiset.retainAll(Collection c)	foreach(o in map) if(o not in c) map.remove(o);	1/2 (50%)	0.61 ±0.01
	ArrayListMultimap.putAll(Object k,Collection c)	foreach(o in c) put(k,o);	8/41 (20%)	0.37 ±0.32
	LinkedHashMultimap.putAll(Object k, Collection c)	foreach(o in c) put(k,o);	0/1 (0%)	0.00 ±0.00
	LinkedHashMultimap.create()	create(100,100)	0/207 (0%)	0.12 ±0.15
	LinkedHashMultimap.create(int,int)	create()	0/202 (0%)	0.12 ±0.15
	LinkedHashMultimap.isEmpty()	size() == 0 ? true : false	0/34 (0%)	0.00 ±0.00
Carrot	ImmutableMultiset.of(Object..c)	foreach(o in c) builder().setCount(o,count(o in c))	13/22 (59%)	0.56 ±0.07
	ImmutableMultiset.of(Object..c)	builder().add(..c).build()	7/19 (37%)	0.24 ±0.12
	ArrayListMultimap.putAll(Object k,Collection c)	foreach(o in c) put(k,o);	1/13 (8%)	0.37 ±0.32
	ImmutableMultiset.of(Object o)	builder().add(o).build()	0/1 (0%)	0.32 ±0.14
	Lists.newArrayList()	new ArrayList()	0/24 (0%)	0.00 ±0.00
	Lists.newArrayList()	new ArrayList(10)	0/24 (0%)	0.00 ±0.00
	Lists.newArrayListWithCapacity(int c)	new ArrayList()	0/20 (0%)	0.00 ±0.00
	Lists.newArrayListWithCapacity(int c)	new ArrayList(c)	0/20 (0%)	0.00 ±0.00
	Maps.newHashMap()	Maps.newHashMapWithExpectedSize(16)	0/54 (0%)	0.00 ±0.00
	Maps.newHashMap()	new HashMap()	0/54 (0%)	0.00 ±0.00
	Maps.newHashMap()	new HashMap(16)	0/54 (0%)	0.00 ±0.00

The most obvious cases are when the two code fragments (C_A and C_B) are either not redundant at all or completely redundant. When there is no redundancy, the equivalence is also completely ineffective to obtain workarounds, and conversely, when we obtain a measure of complete redundancy in the case of Iterators.forArray(a) in Caliper, the equivalence is always effective as a workaround.

The redundancy measure is also a good indicator of the success of a workaround in the other non extreme cases. Consider for example the case of ImmutableMultiset.of(Object..c) in Carrot where the first equivalent alternative has a higher redundancy measure and a higher success ratio than the second one (0.56 ± 0.07 and 0.59 vs. 0.24 ± 0.12 and 0.36). This case shows that the redundancy measure can be an effective predictor to select or rank alternative fragments for use as workarounds.

Overall we obtain a positive correlation (coefficient 0.94) from which we conclude that our redundancy measure is indeed a good indicator and predictor of useful design properties.

D. Threats to Validity

We acknowledge potential problems that might limit the validity of our experimental results. Here we briefly discuss the countermeasures we adopted to mitigate such threats. The internal validity depends on the correctness of our prototype implementations, and may be threatened by the evaluation setting and the execution of the experiments. The prototype tools we used are relatively simple implementations of well defined metrics computed over execution logs and action sequences. We collected and filtered the actions of interests with robust monitoring tools and we carefully tested our implementation with respect to the formal definitions.

Threats to external validity may derive from the selection of case studies. An extensive evaluation of the proposed measurements is out of the scope of this paper, whose goal is to discuss and formally define the concept of software redundancy. We present results obtained on what we would refer to as "ground truth," that is, on cases with clear and obvious expectations that would therefore allow us to check the significance and robustness of the proposed metrics.

V. CONCLUSION

In the past we developed techniques to exploit the redundancy of software, to make software more reliable and adaptive. Several other techniques, more or less mature, exploit the redundancy of software in a similar way. On the basis of this past experience, we now want to gain a deeper and at the same time broader understanding of software redundancy. And the first step is to *model* and *measure* redundancy.

This is what we did in this paper. We formulated a model that we consider expressive and meaningful, and we derived from it a concrete measurement method that we evaluated for its consistency (does the measurement make sense at a very basic level?) and predictive ability (is it a good indicator of useful properties?). Our experiments show that the measurements are indeed consistent and significant, which means that they can be useful in support of a more principled use of redundancy in software design.

We see a number of ways to build upon this work. One would be to enhance the model. The main limitation of the model is that it considers only single-threaded code fragments. Notice in fact that the model, as well as the measure of dissimilarity, is based on the notion of an execution consisting of *one* sequence of actions. One way to model multi-threaded code would be to linearize parallel executions, although that might be an unrealistic oversimplification. Other straightforward extensions include a more extensive experimentation and an improved measurement, in particular in sampling the state space. However, our primary interest is now in using the model and the measurement to study redundancy further. Our ultimate goal is to comprehend redundancy as a phenomenon, to harness its power *by design.*

ACKNOWLEDGMENT

This work was supported by the Swiss National Science Foundation with project *SHADE* (grant n. 200021-138006).

REFERENCES

[1] A. Arcuri and X. Yao. A novel co-evolutionary approach to automatic software bug fixing. In *CEC '08: Proceeding of IEEE Congress on Evolutionary Computation*, pages 162–168, 2008.

[2] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable Secure Computing*, 1(1):11–33, 2004.

[4] A. Carzaniga, A. Gorla, A. Goffi, A. Mattavelli, and M. Pezzè. Cross-checking oracles from intrinsic software redundancy. In *ICSE '14: Proceedings of the 36th International Conference on Software Engineering*, pages 931–942, 2014.

[5] A. Carzaniga, A. Gorla, A. Mattavelli, N. Perino, and M. Pezzè. Automatic recovery from runtime failures. In *ICSE '13: Proceedings of the 35th International Conference on Software Engineering*, pages 782–791, 2013.

[6] A. Carzaniga, A. Gorla, N. Perino, and M. Pezzè. Automatic workarounds for Web applications. In *FSE'10: Proceedings of the 2010 Foundations of Software Engineering conference*, pages 237–246, 2010.

[7] S.-S. Choi and S.-H. Cha. A survey of binary similarity and distance measures. *Journal of Systemics, Cybernetics and Informatics*, pages 43–48, 2010.

[8] W. W. Cohen, P. Ravikumar, and S. E. Fienberg. A comparison of string metrics for matching names and records. In *KDD Workshop on Data Cleaning and Object Consolidation*, pages 73–78, 2003.

[9] A. Diaconescu, A. Mos, and J. Murphy. Automatic performance management in component based software systems. In *ICAC '04: Proceedings of the 1st International Conference on Autonomic Computing*, pages 214–221, 2004.

[10] G. Dobson. Using WS-BPEL to implement software fault tolerance for web services. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 126–133, 2006.

[11] R.-K. Doong and P. G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering and Methodologies*, 3(2):101–130, 1994.

[12] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. Biological Sequence Analysis. Probabilistic Models of Proteins and Nucleic Acids. *Cell Biochemistry and Function*, 17(1):73–73, 1999.

[13] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[14] I. Gashi, P. Popov, V. Stankovic, and L. Strigini. On designing dependable services with diverse off-the-shelf SQL servers. In *Architecting Dependable Systems II*, volume 3069 of *Lecture Notes in Computer Science*, pages 191–214. Springer, 2004.

[15] A. Goffi, A. Gorla, A. Mattavelli, M. Pezzè, and P. Tonella. Search-based synthesis of equivalent method sequences. In *FSE'14: Proceedings of the 22nd International Symposium on the Foundations of Software Engineering*, pages 366–376, 2014.

[16] A. Gotlieb. Exploiting symmetries to test programs. In *ISSRE '03: Proceedings of the 14th International Symposium on Software Reliability Engineering*, pages 365–374, 2003.

[17] M. Hennessy and R. Milner. On observing nondeterminism and concurrency. In J. de Bakker and J. van Leeuwen, editors, *Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 299–309. Springer, 1980.

[18] Y. Higo and S. Kusumoto. How Should We Measure Functional Sameness from Program Source Code? An Exploratory Study on Java Methods. In *FSE'14: Proceedings of the 22nd International Symposium on Foundations of Software Engineering*, pages 294–305, 2014.

[19] M. A. Jaro. Probabilistic linkage of large public health data files. *Statistics in Medicine*, 14(5-7):491–498, 1995.

[20] L. Jiang and Z. Su. Automatic mining of functionally equivalent code fragments via random testing. In *ISSTA '09: Proceedings of the 18th International Symposium on Software Testing and Analysis*, pages 81–92, 2009.

[21] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12:96–109, 1986.

[22] N. Looker, M. Munro, and J. Xu. Increasing web service dependability through consensus voting. In *COMPSAC '05: Proceedings of the 29th International Computer Software and Applications Conference*, pages 66–69, 2005.

[23] L. Marek, A. Villazón, Y. Zheng, D. Ansaloni, W. Binder, and Z. Qi. Disl: A domain-specific language for bytecode instrumentation. In *AOSD'12: Proceedings of the 11th International Conference on Aspect-oriented Software Development*, pages 239–250, 2012.

[24] A. Mosincat and W. Binder. Transparent runtime adaptability for BPEL processes. In A. Bouguettaya, I. Krüger, and T. Margaria, editors, *ICSOC '08: Proceedings of the 6th International Conference on Service Oriented Computing*, volume 5364 of *Lecture Notes in Computer Science*, pages 241–255, 2008.

[25] H. Naccache and G. Gannod. A self-healing framework for web services. In *ICWS '07: Proceedings of the 2007 IEEE International Conference on Web Services*, pages 398–345, 2007.

[26] G. Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[27] R. D. Nicola and M. Hennessy. Testing equivalences for processes. *Theoretical Computer Science*, 34(1-2):83 – 133, 1984.

[28] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[29] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for java. In *OOPSLA'07: Proceedings the 22nd Conference on Object-Oriented Programming Systems and Applications*, pages 815–816, 2007.

[30] B. Randell. System structure for software fault tolerance. *SIGPLAN Notes*, 10(6):437–449, 1975.

[31] S. M. Sadjadi and P. K. McKinley. Using transparent shaping and Web services to support self-management of composite systems. In *ICAC '05: Proceedings of the 2nd International Conference on Automatic Computing*, pages 76–87, 2005.

[32] P. Sellers. On the theory and computation of evolutionary distances. *SIAM Journal on Applied Mathematics*, 26(4):787–793, 1974.

[33] S. Subramanian, P. Thiran, N. C. Narendra, G. K. Mostefaoui, and Z. Maamar. On the enhancement of BPEL engines for self-healing composite web services. In *SAINT '08: Proceedings of the 2008 International Symposium on Applications and the Internet*, pages 33–39, 2008.

[34] Y. Taher, D. Benslimane, M.-C. Fauvet, and Z. Maamar. Towards an approach for Web services substitution. In *IDEAS '06: Proceedings of the 10th International Database Engineering and Applications Symposium*, pages 166–173, 2006.

[35] W. Weimer, a. C. L. G. ThanVu Nguyen, and S. Forrest. Automatically finding patches using genetic programming. In *ICSE'09: Proceeding of the 31st International Conference on Software Engineering*, pages 364–374, 2009.

[36] S. S. Yau and R. C. Cheung. Design of self-checking software. In *Proceedings of the International Conference on Reliable Software*, pages 450–455, 1975.