Intrinsic Redundancy for Reliability and Beyond

Alberto Goffi, Alessandra Gorla, Andrea Mattavelli, and Mauro Pezzè

Abstract Software redundancy is an essential mechanism in engineering. Different forms of redundant design are the core technology of well-established reliability and fault tolerant mechanisms in traditional engineering as well as in software engineering. In this paper we discuss *intrinsic software redundancy*, a type of redundancy that is not added explicitly at design time to improve runtime reliability, but is natively present in modern software system due to independent design and development decisions. We introduce the concept of intrinsic redundancy, discuss its diffusion and the reasons for its presence in modern software systems, indicate how it can be automatically identified, and present some current and future applications of such form of redundancy to produce more reliable software systems at affordable costs.

1 Introduction

Reliability, which is *the ability of a system or component to perform its required functions under stated conditions for a specified period of time* [1], is a key property of engineered products and in particular of software artifacts. Safety critical applications must meet the high reliability standards required for their field deployment, time- and business-critical applications must obey strong reliability requirements, everyday

Alberto Goffi
USI Università della Svizzera italiana, Switzerland, e-mail: alberto.goffi@usi.ch

Alessandra Gorla
IMDEA Software Institute, Spain, e-mail: alessandra.gorla@imdea.org

Andrea Mattavelli
Imperial College London, United Kingdom, e-mail: a.mattavelli@imperial.ac.uk

Mauro Pezzè
USI Università della Svizzera italiana, Switzerland and University of Milano-Bicocca, Italy, e-mail: mauro.pezze@usi.ch

and commodity products shall meet less stringent, but still demanding customer requirements.

Reliability exploits runtime mechanisms that prevent or mask failures by relying on some form of *redundancy*: *redundant implementation* that is based on runtime replicas of the same or similar components to temper the failure of one or more elements; *redundant design* that relies on independently designed components with the same functionality to reduce the impact of faults; *redundant information* that replicates information to compensate for corrupted data.

In classic engineering practice, examples of redundant *implementations* are the third engine of the McDonnell Douglas DC-10 and MD-11 aircraft, added in the late sixties to tolerate single engine failures,[1] The Boeing 777 control system which is compiled with three different compilers and runs on three distinct processors [49], the design of Systems-on-a-Chip (SoC), which relies on redundant components,[2] the Redundant Array of Independent Disks (RAID) technology that combines multiple physical disk drive components into a single logical unit [36]. An example of redundant *design* in classic engineering is the redundant design practice for building bridges to prevent localized failures from propagating to the whole bridge structure [19, 20, 32]. A well-known instance of redundant *information* is the Hadoop Distributed File System (HDFS) that improves reliability by replicating data [41].

In software engineering, redundant design is a basic principle of fault-tolerant approaches, which explicitly add redundancy to tolerate unavoidable faults [43]. Examples of redundant software design are N-version programming modules [13], rollback and recovery techniques, error-correcting codes and recovery blocks [23, 28, 37, 38]. Redundant design relies on elements *explicitly* added to the system, which come with extra costs that limit their applicability.

In this chapter, we discuss a form of redundancy that is present in software systems independently from reliability issues and that can be exploited with negligible additional costs to improve software reliability. We refer to such form of redundancy as *intrinsic* software redundancy. Figure 1 shows a simple example of redundant methods in class `java.util.Stack`: the invocations of methods `clear()`, `removeAllElements()`, and `setSize(0)` produce the same result (an empty stack) by executing the different code fragments reported in the figure. The different albeit equivalent[3] methods derive from design choices that are independent from reliability issues, and as such we refer to them as *intrinsically redundant methods*.

Some recent studies indicate that software systems present many forms of *intrinsic* redundancy that is a form of redundancy that derives from design and implementation decisions that go beyond the explicit choice of adding redundant elements for the sake of reliability [3, 6, 7, 8, 10, 11, 12, 18, 25, 29, 42, 44]. Intrinsic redundancy has been successfully exploited to improve software reliability [42], build self-healing systems [3, 7, 8, 10, 44], and generate test oracles [6].

[1] https://federalregister.gov/a/07-704

[2] http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=43464

[3] Here and in the whole chapter, we use the term *equivalent method* to indicate methods that produce results indistinguishable from an external viewpoint, as discussed in details in Section 2.

```
public void clear() {
    removeAllElements();
}

public synchronized void removeAllElements() {
  modCount++;
  // Let gc do its work
  for (int i = 0; i < elementCount; i++)
    elementData[i] = null;
  elementCount = 0;
}

public synchronized void setSize(int newSize) {
  modCount++;
  if (newSize > elementCount) {
    ensureCapacityHelper(newSize);
  } else {
    for (int i = newSize ; i < elementCount ; i++) {
      elementData[i] = null;
    }
  }
  elementCount = newSize;
}
```

Fig. 1 Sample redundant methods from class `java.util.Stack`

We discuss the concept of intrinsic redundancy in software systems, argue about the reasons for its presence, and provide data about the scale of the phenomenon in Section 2. We investigate the problem of automatically uncovering intrinsic software redundancy, and present a search-based approach to automatically identify redundancy intrinsically present at the method call sequence level in Section 3. We illustrate how to exploit the redundancy that is intrinsically present in software systems for producing self-healing systems in Section 4 and for generating semantically relevant test oracles in Section 5. We identify challenging research problems that may be effectively addressed by exploiting intrinsic software redundancy, focusing in particular on the areas of performance optimization and security in Section 6.

2 Intrinsic Software Redundancy

Two *different* executions are *redundant* if they produce *indistinguishable functional results* from an external viewpoint. For example methods `clear()`, `removeAllElements()` and `setSize(0)` of class `java.util.Stack` in Figure 1 execute different statements and run with different execution time, but produce the same visible functional result (an empty stack) and are thus *redundant*. Redundant executions may differ in the sequence of executed actions as in the case of methods `clear()`,

```
map.setCenter(new GLatLng(37, −122), 15)
// is equivalent to
setTimeout(”map.setCenter(new GLatLng(37, −122), 15)”, 500)

map = new GMap2(document.getElementById(”map”));
setTimeout(”map.setCenter(new GLatLng(37, −122), 15)”, 500);
map.openInfoWindow(new GLatLng(37.4, −122), ”Hello World!”);
```

Fig. 2 Sample redundant method calls in JavaScript in the presence of multithread clients

`removeAllElements()` and `setSize(0)`, or simply in the execution order of same shared actions, as in the case of the code of Figure 2, where `setTimeout` does not alter the sequence of executed actions, but may result in different order of execution with multithreaded clients.[4]

Redundancy is present in software systems at all abstraction levels, from single bits to entire software systems. The Cyclic Redundancy Check (CRC) is an example of redundancy at the bit level. Methods of the same or different classes that produce the same results, like methods `clear()`, `removeAllElements()` and `setSize(0)`, are examples of redundancy at the method call sequence level. Libraries and services with overlapping functionality, for instance the log4J[5] library and the standard Java class `java.util.Logging` that largely overlap, are examples of redundancy at the subsystem and system level.

In this paper we discuss intrinsic redundancy of software systems referring to the *method call sequence* level:

- we refer to deterministic object-oriented software systems at the method call level; the interested reader can find a generalization of the concept of intrinsic redundancy to non-deterministic systems in Mattavelli's PhD thesis [33];
- we focus on redundancy intrinsically present in the systems due to independent design or implementation decisions, and not in redundancy explicitly added at design time as a first-class design choice like in the case of N-version programming;
- we consider only externally observable functional behavior, ignoring other aspects of the executions such as differences in the internal state, data structures, execution time and performance.

Two methods are redundant if they differ in at least one execution trace and produce indistinguishable results for all executions. Two execution traces are different if they differ in at least one event or in their order. Two methods produce indistinguishable results if their executions compute the same results and lead to equivalent states from an external observer's viewpoint. More precisely, the executions of two methods m_1 and m_2 of a class C with inputs i_1 and i_2 that produce outputs o_1 and o_2

[4] The insertion of a `setTimeout` is frequently used as workaround for issues in Google Maps.

[5] http://logging.apache.org/log4j

Table 1 Redundant method call sequences in open-source Java systems

System	Classes	Redundant methods	Avg. per class
Apache Ant	213	804	3.80
Apache Lang3	1	45	45.00
Apache Lucene	160	205	1.28
Apache Primitives	16	216	13.50
Canova	95	345	3.63
CERN Colt	27	380	14.07
Eclipse SWT	252	1494	5.93
Google Guava	116	1715	14.78
GraphStream	9	132	14.67
Oracle JDK	2	85	42.50
Joda-Time	12	135	11.25
Trove4J	54	257	4.76
Total	957	5813	6.07

and reach states s_1 and s_2, respectively, are equivalent if $o_1 = o_2$ and no sequence of calls of methods of class C executed from s_1 and s_2 produces different results.

Checking for the diversity of two methods is easy, since we only need to compare the execution traces and find two different ones; checking for their equivalence is complex, since it implies demonstrating that all possible executions produce states that are indistinguishable with any interaction sequence. In Section 3, we present an automatic approach to infer the *likely* equivalence of method call sequences, which is based on heuristics, and as such is an imprecise albeit practical and useful approximation of equivalence.

Intrinsic redundancy may stem from many design and commercial practices, including but not limited to design for reusability, performance optimization, backward compatibility, and lack of software reuse. Reusable software systems, and in particular libraries, provide standard application programming interfaces (APIs) that emphasize flexibility over conciseness. For instance, the popular JQuery library[6] provides many alternative methods to display elements in a Web page: `show()`, `animate()`, `fadeTo()`, `fadeIn()`. Different albeit observationally equivalent functionalities are often present due to performance optimizations. For instance the GNU Standard C++ Library implements the basic stable sorting function using the insertion-sort algorithm for small inputs and merge-sort for the general case. Many libraries continue to offer legacy code to guarantee backward compatibility. For instance, the Java 8 Class Library contains dozens of deprecated classes and hundreds of deprecated methods that overlap with the functionality of newer classes and methods.[7] Time pressure and cost factors reduce the effectiveness of inter- and intra-project communications and limit the degree of reuse. Often developers are simply not aware that a functionality is already available in the system, and implement the same functionality multiple times [5, 30].

[6] http://jquery.com

[7] http://docs.oracle.com/javase/8/docs/api/deprecated-list.html

Intrinsic redundancy is surprisingly widespread in many software systems. Table 1 summarizes the results of our empirical analysis of the presence of intrinsic redundancy that derives from good design for reusability practice in a set of open-source Java libraries at the intra-class method call sequence level. Table 1 indicates a large amount of redundant methods (column *Redundant methods*) within the examined classes (column *Classes*), which in turn lead to a considerable quantity of redundant methods within each class (column *Avg. per class*). Table 1 reports few summary data, interested readers can find additional details in [6, 7, 8, 10, 12].

3 Mining Software Redundancy

Identifying redundant method call sequences by manually inspecting the software systems is an error prone and effort demanding activity. This section suggests that the intrinsic redundancy of software systems can be automatically identified by an approach, *Search-Based Equivalent Synthesis (SBES)*, that automatically detects redundant method call sequences in Java classes [22, 34].

Given a target method of a Java class, *SBES* synthesizes sequences of method calls that are redundant to the target method, that is, sequences of method calls that produce results that are indistinguishable from the results of the target method while executing different actions. *SBES* approximates the equivalence of method call sequences referring to a finite set of *execution scenarios*. We refer to the method call sequences that *SBES* identifies as *likely equivalent sequences*, since they are proven equivalent to the target method for the considered finite set of execution scenarios, but may differ for other unforeseen executions.

Given a method *m* of a Java class *C*, *SBES* incrementally synthesizes candidate redundant method call sequences to *m*. For each candidate sequence, *SBES* then explores the input space of the candidate sequence, looking for inputs that produce results different from *m*. If such inputs are found, *SBES* discards the candidate sequence and proceeds with a new candidate. Otherwise, if no input that distinguishes the candidate sequence from *m* is found before a given timeout, *SBES* deems the sequence as a *likely* redundant method call sequence of *m*.

We illustrate *SBES* referring to method `pop()` of the `java.util.Stack` class. *SBES* starts with an initial scenario that consists of a set of randomly selected test cases for the target method, for instance the simple test case `test01` at lines 1–6 in Figure 3. It then looks for a sequence of method calls that produces indistinguishable results with respect to the candidate method for the current scenario. It does so by exploiting search-based algorithms, and in particular the genetic algorithms implemented in EvoSuite [17]. In the example, *SBES* synthesizes the candidate sequence of method calls at lines 8–12 in Figure 3, which produces the same result of the target method `pop()` for the initial scenario.

SBES validates the candidate by looking for inputs that distinguish the candidate method call sequence from the target method, by exploiting again the genetic algorithms implemented in EvoSuite. In the example, *SBES* finds the counterexample

```
 1 // Initial execution scenario
 2 public void test01() {
 3   Stack<Integer> s = new Stack<>();
 4   s.push(1);
 5   Integer result = s.pop();
 6 }

 8 // First candidate equivalent method call sequence:
 9 // stack.remove(0) candidate equivalent to stack.pop()
10 Stack<Integer> s = new Stack<>();
11 s.push(1);
12 Integer result = s.remove(0);

14 // Counterexample
15 Stack<Integer> s = new Stack<>();
16 s.push(2);
17 s.push(1);
18 Integer result = s.pop();

20 // Second candidate equivalent method call sequence:
21 // stack.remove(stack.size() − 1) candidate equivalent to stack.pop()
22 Stack<Integer> s = new Stack<>();
23 int x0 = s.size();
24 int x1 = x0 − 1;
25 Integer result = s.remove(x1)
```

Fig. 3 Candidate method call sequences, counterexamples and execution scenarios for method `pop()` of class `java.util.Stack`

shown at lines 14–18 in Figure 3, that is, an input that differentiates the results produced by the candidate method call sequence and the target method. If *SBES* finds a counterexample, as in this case, it discards the current candidate method call sequence, adds the counterexample to the current execution scenario, and iterates, looking for a new candidate method call sequence indistinguishable from the target method for the new scenario. By incrementally adding the counterexamples to the execution scenarios, *SBES* restricts the search to a smaller set of potential candidates, thus improving the likelihood of generating method call sequences that are redundant with respect to the target method.

In the example, *SBES* synthesizes the new candidate at line 20–25 in Figure 3. The search for inputs that differentiate the candidate method call sequence from the target method fails in identifying a counterexample with a timeout. Thus *SBES* returns the synthesized sequence as likely redundant to the target method. In the example, the synthesized sequence is indeed redundant, since it produces results that are indistinguishable from the target method for every possible input. Our experiments confirm that most sequences that *SBES* synthesizes as likely redundant are often redundant indeed. In general, *SBES* may iterate several times before finding a likely redundant sequence. *SBES* may also fail in synthesizing a new candidate, and in this

Table 2 Effectiveness of SBES

System	Class	Redundant	Redundant Methods
		Methods	Found by SBES
Oracle JDK	Stack	45	32 (71%)
Graphstream	Path	5	5 (100%)
	Edge	20	20 (100%)
	SingleNode	12	12 (100%)
	MultiNode	12	12 (100%)
	Vector2	21	21 (100%)
	Vector3	22	22 (100%)
Google Guava	ArrayListMultimap	18	12 (67%)
	ConcurrentHashMultiset	16	6 (38%)
	HashBasedTable	13	2 (15%)
	HashMultimap	13	13 (100%)
	HashMultiset	19	19 (100%)
	ImmutableListMultimap	20	2 (10%)
	ImmutableMultiset	20	3 (15%)
	LinkedHashMultimap	13	12 (92%)
	LinkedHashMultiset	19	19 (100%)
	LinkedListMultimap	17	11 (65%)
	Lists	16	15 (94%)
	Maps	12	8 (67%)
	Sets	25	21 (84%)
	TreeBasedTable	17	3 (18%)
	TreeMultimap	12	8 (67%)
	TreeMultiset	34	34 (100%)
Total		421	312 (74%)

case we terminate the synthesis process and return the set of likely redundant method call sequences found.

Table 2 summarizes the experimental data reported in Goffi et al. [22] and Mattavelli et al. [34] that confirm the effectiveness of SBES in automatically identifying redundant method call sequences. The table reports the number of redundant method call sequences in the considered classes (column *Class*) as found by manually inspecting the code (column *Redundant Methods*) and the amount and percentage of automatically identified redundant method call sequences (column *Redundant Methods Found by SBES*). As reported in the table, *SBES* can find a large amount of redundant method sequences with an average of 74% and a median over 88%, and fails in identifying most of the missing redundancies for technological limitations of the current prototype implementation that does not satisfactorily deal with subtle use of some Java constructs.

4 Runtime Failure Recovery

Intrinsic redundancy is exploited in many ways to relieve the effects of faulty code fragments at different abstraction levels, from single code statements to entire components. For instance, many approaches exploit redundancy at the service component level to overcome failures caused by either malfunctioning services or unforeseen changes in the functionality offered by the current reference implementation [3, 39, 45]. Other approaches exploit some form of intrinsic redundancy to automatically patch faulty code at the statement level, for instance LeGoues et al. [31] and Arcuri et al. [2] use genetic programming to automatically fix faults, while Sidiroglou-Douskos et al. make use of code fragments that are extracted from "donor" applications [42]. Automatic *runtime* code repair techniques patch the code at runtime to mitigate the effect of failures during the software execution.

In this section we illustrate the use of intrinsic software redundancy to recover from runtime failures by referring to the *Automatic Workaround Approach (AWA)*, which exploits intrinsic redundancy at the method call sequence level to automatically recover from failures at runtime [7, 8, 10]. A *workaround* substitutes a faulty code fragment with a different redundant code fragment that produces the same intended behavior while executing different code that avoids the faulty operations. For example the redundant method call `setTimeout(map.setCenter(new GLatLng(37,122),15),500)` shown in Figure 2 can be successfully exploited as a workaround for `map.setCenter(new GLatLng(37,122),15)` to solve the now closed issue 519 of the Google Maps API.[8]

In a nutshell, *AWA* detects a failure, rolls back to a consistent state, substitutes the faulty code fragment with a redundant code fragment, and executes the new code. The *AWA* key ingredients are: (i) a *failure detection* mechanism that reveals failures at runtime, (ii) a *save & restore* mechanism that rolls back the application to a consistent state after a failure, and (iii) a *healing engine* that replaces the failing code fragment with a redundant fragment.

When dealing with Web applications, *AWA* focuses on JavaScript libraries and relies on the stateless nature of classic Web applications to ensure state consistency. When dealing with Java applications, *AWA* augments the core healing-engine with mechanisms to detect failures, and save and restore the state. In the next paragraphs, we briefly summarize the three main *AWA* ingredients for both Web and Java applications. The interested readers can refer to [7] and [8, 10] for details on *AWA* for Java programs and Web applications, respectively. We illustrate the approach referring to *AWA* successfully exploiting the redundant method calls shown in Figure 2 to heal the now closed issue 519 of Google Maps API as a running example.

Failure Detection

The *AWA* failure detection mechanism reveals failures and triggers the *AWA* healing engine at runtime. When dealing with Web applications, *AWA* takes advantage of the interactive nature of the application, and relies on users who are given an intuitive way, for instance a browser extension, both to signal undesired outputs and to validate the behavior of the application after a potential workaround is applied. When dealing

[8] https://code.google.com/p/gmaps-api-issues/

with Java applications, *AWA* relies on implicit failure detectors such as runtime exceptions and violations of pre-/post-conditions and invariants. In our experiments with Web applications we provided users with a button to signal failures.

In the running example, users signal the lack of the expected pop up window for additional information about the location on the map through a *Fix me* button that we added to Google Chrome [9].

Save & Restore

The *AWA* save and restore mechanism incrementally saves intermediate execution states to roll back to a consistent state, that is, a state before the occurrence of a failure. When dealing with Web applications, *AWA* takes advantage of the stateless nature of the client side, assuming that the JavaScript code executed on the client side implements stateless components, and simply reloads the page without worrying about possible side-effects on the state of the application. When dealing with Java applications, the save and restore mechanism periodically saves the execution states, and must find a good compromise for the frequency. Saving operations should not be too frequent, to limit the overhead, and not too sporadic either, since they may also cover I/O operations that may be difficult or impossible to restore. *AWA* identifies code regions that include redundant code—and that can thus be fixed with automatic workarounds—and saves the state before executing these regions. In principle, these code regions may extend over sections of the application at any level of granularity, in practice they usually extend within a method body.

Healing Engine

The *AWA* healing engine executes code that is redundant with respect to the code fragment likely responsible for the detected failure, aiming to restore a correct execution. In general, the *AWA* healing engine iteratively restores the state of the application to a previously saved checkpoint and executes a code fragment which is redundant with respect to the code that is a suspect responsible of the failure, until either the failure does not occur or the available redundant fragments are exhausted. If the failure does not occur after the healing action, *AWA* successfully prevents the failure and the execution of the application proceeds as if no failure occurred. If the failure persists, *AWA* cannot prevent the failure and forwards the failure to the application. In the presence of multiple alternatives, *AWA* selects the alternative candidates by relying on heuristics based on the past success of the redundant alternatives.

When dealing with Web applications, *AWA* simply extracts the JavaScript code from the failing page, replaces the suspect code with a redundant code fragment, and displays the new page to the user, who can either continue interacting with the application or signal the persistency of a problem. In this latter case, *AWA* iterates with a new redundant code fragment.

In the running example, the failing page contains several statements with known redundant method call sequences. After two failing attempts, where *AWA* substitutes a statement in the page with a redundant one without solving the failure and triggering new user's *Fix me* requests, *AWA* substitutes statement `map.setCenter(new GLatLng(37, 122), 15)` with the candidate workaround shown in Figure 2, and reloads the page successfully healing the failure [10].

AWA assumes the availability of the set of redundant alternatives present in the target application, that can be automatically identified with the search-based approach presented in Section 3, and pre-processes the application off-line to enable the on-line healing mechanism. It analyzes the application *off-line* to locate code fragments with redundant alternatives, pre-compiles all the redundant code fragments, and instruments the application with the necessary code to select those alternative redundant fragments at runtime in response to a failure. At runtime, *AWA* saves the state of the application at the identified checkpoints, and reacts to failures by executing workarounds.

The experimental data reported in [7, 8, 10] and collected on three popular Web libraries (Google Maps, JQuery, and Youtube)[9] and four Java applications (Fb2pdf, Caliper, Carrot2, and Closure compiler)[10] that use two popular Java libraries (Google Guava and JodaTime)[11] indicate that *AWA* is indeed effective: It automatically applies workarounds for 100 out of 146 known faults for the considered Web applications, and for a percentage that varies between 19% and 48% of the failure-inducing faults in the considered Java applications.

5 Automated Oracles

Software testing and in particular automatic generation of test oracles is another important area where redundancy finds interesting applications. Test oracles check the results of the code execution and signal discrepancies between actual and expected behavior [4]. Their efficacy and cost play a key role in cost-effective test automation approaches. Manual test case generation produces effective oracles, but is very expensive and strongly impacts on the cost of testing. Generating useful test oracles automatically is extremely valuable, but is generally difficult and in some cases may not be practical or even possible [47].

In her seminal work, Weyuker proposes *pseudo-oracles* that exploit explicit redundancy given in the form of multiple versions of a system to check program results [47]. Doong and Frankl define the ASTOOT approach that relies on redundancy that transpires from algebraic specifications to automatically generate test

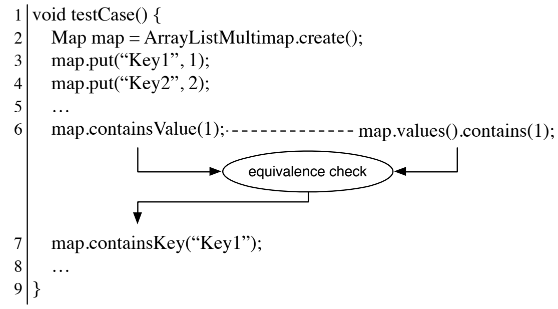
[9] Google Maps (http://code.google.com/apis/maps), JQuery (http://jquery.com), Youtube (http://code.google.com/apis/youtube)

[10] Caliper (https://github.com/google/caliper), Carrot2 (http://project.carrot2.org), Closure Compiler (https://github.com/google/closure-compiler), Fb2pdf (http://fb2pdf.com)

[11] Guava (https://github.com/google/guava), JodaTime (https://github.com/JodaOrg/joda-time)

inputs and oracles [16]. The metamorphic testing approach introduced by Chen et al. almost a decade later exploits a form of redundant information given as metamorphic relations to automatically generate test oracles [14].

In this section, we illustrate the role of intrinsic redundancy in automated software testing by means of *cross-checking oracles*, which exploit the intrinsic redundancy at the method call sequence level to automatically generate application specific oracles [6]. Figure 4 illustrates the cross-checking oracle approach by referring to the invocation of method `containsValue(value)` of the class `ArrayListMultimap`. Once mined the redundancy between methods (methods `map.containsValue(value)` and `map.values().contains(value)` in the example), cross-checking oracles complement each invocation of a method for which we know that there exists some redundancy (`map.containsValue(value)`) with a parallel invocation of the redundant method (`map.values().contains(value)`) followed by a comparison of the produced results and reached states (`equivalence check` in the figure). The oracle signals a problem if two redundant methods invoked in the same context either produce different results or reach states that are distinct from an external observer viewpoint.



```
void testCase() {
   Map map = ArrayListMultimap.create();
   map.put(“Key1”, 1);
   map.put(“Key2”, 2);
   …
   map.containsValue(1);              map.values().contains(1);

   map.containsKey(“Key1”);
   …
}
```

Fig. 4 A visual representation of a cross-checking oracle [21]

Cross-checking oracles can be generated and executed automatically given a set of redundant code elements, and provide a way of checking for faults that depend on the semantics of the program. They are automatically deployed into test suites through binary instrumentation, and rely on a deep-clone mechanism to ensure a reasonable level of isolation between the executions of redundant methods, with a limited execution overhead. Cross-checking oracles implement a finite approximation of equivalence that checks the equality of both the externally visible results and the states reached after executing the redundant methods, through the concatenation of a finite sequence of method invocations.

The experimental results reported in [6], indicate that cross-checking oracles substantially improve the effectiveness of automatically generated test suites that rely on implicit oracles, and in some cases can also improve specific oracles written by the developers.

6 Beyond Functional Intrinsic Redundancy

In the previous sections, we illustrated the application of intrinsic software redundancy in the context of software reliability, and in particular for the design of mechanisms for runtime failure recovery and automated oracles, focusing on functional properties. The notion of intrinsic software redundancy can be extended to *non-functional* properties, and find many new applications. In this section we identify future research directions towards applications of non-functional software redundancy in new contexts, namely performance optimization and security.

Performance Optimization

Redundant code fragments execute different sequences of actions that may lead to notable differences in runtime behavior and resource usage. Such differences can be exploited to alleviate performance as well as resource consumption problems, depending on the operative conditions. For example, mobile devices offer several connectivity options that span from mobile protocols, WiFi connectivity, Bluetooth access points and so on. The optimal choice of connectivity depends on the operational conditions and on trade-off between performance, urgency, battery consumption, privacy, security that cannot be predicted and efficiently wired in ad design time.

Recent work has investigated the use of various forms of redundancy for improving non-functional properties. The GISMOE approach exploits genetic programming to generate program variants to address different non-functional objectives [24]. The competitive parallel execution approach (CPE) increases the overall system performance by executing multiple variants of the same program in parallel [46]. Self-adaptive containers minimize the runtime costs by monitoring the runtime performance of the application and automatically selecting the best internal data structures [27]. Misailovic et al. propose a new profiler to identify computations that can be replaced with alternative—and potentially less accurate—computations that provide better performance [35]. The applicability and effectiveness of the different approaches is bounded by the techniques used to identify and exploit redundancy and the kind of redundancy that they infer and exploit.

The redundancy intrinsically present in software systems offers new opportunities for automatically improving performance and resource consumption at runtime. The key idea is to devise a “profile” of the redundant code that captures non-functional differences among the alternatives, for instance in terms of timing, memory or battery consumption, or network utilization. This *non-functional profile* can be updated and exploited at runtime, while efficiently monitoring the system execution, to adapt the behavior to meet, or improve, performance and resource utilization requirements.

For example, the non-functional differences of redundant video streaming algorithms, such as runtime performance, battery consumption and network utilization, can be exploited to face performance problems due to unpredictable environment changes.

```
/**
 * Breaks a path up into a Vector of path elements, tokenizing on File.separator.
 * @param path Path to tokenize. Must not be null.
 * @return a Vector of path elements from the tokenized path
 */
public static Vector tokenizePath(String path) {...}

/**
 * Same as tokenizePath but faster.
 */
public static String[] tokenizePathAsArray(String path) {...}
```

Fig. 5 Documentation of the tokenizePath and tokenizePathAsArray methods in Apache Ant.

Figure 5 illustrates the approach with a pair of redundant *tokenize* methods, which explicitly offer different runtime performance. The two methods perform differently depending on operational conditions, like the frequency of invocations on the same or similar arrays, the dimension and the content of the arrays and so on, and coexist in the Apache Ant library to offer different design opportunities. They can be mutually exchanged based on the monitored operational profile and the discrepancies between actual and expected performance. The non-functional profile shall capture the various performance profiles of the two methods, identify the situations that may impact on performance differences, and in general the non-functional differences that may suggest the use of one of the two methods depending on the runtime conditions.

Security

Redundant code fragments may provide different security levels that can also be exploited to tackle security issues and overcome runtime problems. Recent work has investigated the possibility of exploiting some form of explicit redundancy to mitigate security issues. *N-variant systems* increase application security by executing different synthesized variants of the same program in parallel [15]. *Orchestra* tackles security issues by creating multiple variants of the same program based on various compiler optimizations [40]. *Replicated browsers* tackles security problems by executing different browsers in parallel [48].

Redundant code fragments offer a promising alternative to implement new security mechanisms by defining a security-profile of redundant code fragments and by efficiently executing the various alternatives to identify divergences in their runtime behavior, for instance with a multi-version execution framework [26].

Figure 6 shows an example of redundant code fragments that can be exploited to improve security. Both methods `gets` and `scanf` can be successfully exploited by attackers through buffer overflows when invoked with not well-terminated strings. Method `fgets` provide the same functionality of `gets` and `scanf`, but prevents buffer

```
// Reads characters from the standard input (stdin) and stores them as a C string
// into str until a newline character or the end-of-file is reached.
  char * gets (char *str);

// Reads data from stdin and stores them according to the parameter format into
// the locations pointed by the additional arguments.
  int scanf (const char *format, ...);

// Reads characters from stream and stores them as a C string into str until
// (num-1) characters have been read or either a newline or the end-of-file is
// reached, whichever happens first.
  char * fgets (char *str, int num, FILE *stream);
```

Fig. 6 Documentation of the gets, fgets and scanf methods of the C Standard Library.

overflows. The information about the redundancy of these three methods provides the necessary knowledge to develop mechanisms to prevent security threats.

7 Conclusions

Redundancy is a traditional ingredient of many mechanisms for improving reliability and fault tolerance at runtime. Classic engineering approaches rely on different forms of redundancy explicitly added at design time, and suitably exploited at runtime. Such form of redundancy may be expensive to produce, and may be relegated to systems whose reliability requirements balance the extra costs of adding redundancy explicitly, as in the case of N-version programming for safety critical applications.

Recent studies have identified a different form of redundancy that is not explicitly added at design time for improving reliability, but is present for independent design and development decisions, and that we refer to as *intrinsic software redundancy*.

In this paper we summarize the recent advances in the study and exploitation of intrinsic software redundancy, and we indicate promising research directions. We define intrinsic software redundancy informally, discuss the source of such kind of redundancy, and show its presence in relevant software applications. We present an approach to automatically identify intrinsic software redundancy at the method call sequence level, thus providing evidence of the limited costs of gathering information about redundant code elements at a convenient abstraction level.

We report some applications of intrinsic software redundancy to improve reliability at runtime, by proposing the automatic generation of runtime workarounds and program specific oracles. We conclude by indicating new relevant domains that can benefit from the presence of intrinsic redundancy in software systems.

8 Acknowledgments

This work was supported in part by the Swiss National Science Foundation with projects *SHADE* (grant n. 200021-138006), *ReSpec* (grant n. 200021-146607), *WASH* (grant n. 200020-124918), and *SHADE* (grant n. 200021-138006), by the European Union FP7-PEOPLE-COFUND project *AMAROUT II* (grant n. 291803), by the Spanish Ministry of Economy project *DEDETIS*, and by the Madrid Regional Government project *N-Greens Software* (grant n. S2013/ICE-2731).

References

1. IEEE Recommended Practice on Software Reliability (2008)
2. Arcuri, A., Yao, X.: A novel co-evolutionary approach to automatic software bug fixing. In: Proceedings of IEEE Congress on Evolutionary Computation, CEC '08, pp. 162–168. IEEE Computer Society (2008)
3. Baresi, L., Guinea, S., Pasquale, L.: Self-healing BPEL processes with dynamo and the jboss rule engine. In: International Workshop on Engineering of Software Services for Pervasive Environments, ESSPE '07, pp. 11–20 (2007)
4. Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S.: The oracle problem in software testing: A survey. IEEE Transactions on Software Engineering **41**(5), 507–525 (2015)
5. Bauer, V., Eckhardt, J., Hauptmann, B., Klimek, M.: An exploratory study on reuse at google. In: Proceedings of the 1st International Workshop on Software Engineering Research and Industrial Practices, SER&IPs 2014, pp. 14–23. ACM (2014)
6. Carzaniga, A., Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M.: Cross-checking oracles from intrinsic software redundancy. In: Proceedings of the International Conference on Software Engineering, ICSE '14, pp. 931–942. ACM (2014)
7. Carzaniga, A., Gorla, A., Mattavelli, A., Pezzè, M., Perino, N.: Automatic recovery from runtime failures. In: Proceedings of the International Conference on Software Engineering, ICSE '13, pp. 782–791. IEEE Computer Society (2013)
8. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds for web applications. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pp. 237–246. ACM (2010)
9. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: RAW: Runtime automatic workarounds. In: ICSE '10: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (Tool Demo), pp. 321–322. ACM (2010)
10. Carzaniga, A., Gorla, A., Perino, N., Pezzè, M.: Automatic workarounds: Exploiting the intrinsic redundancy of web applications. ACM Transactions on Software Engineering and Methodologies **24**(3), 16 (2015)
11. Carzaniga, A., Gorla, A., Pezzè, M.: Fault handling with software redundancy. In: R. de Lemos, J. Fabre, C. Gacek, F. Gadducci, M. ter Beek (eds.) Architecting Dependable Systems VI, pp. 148–171. Springer (2009)
12. Carzaniga, A., Mattavelli, A., Pezzè, M.: Measuring software redundancy. In: Proceedings of the 37th International Conference on Software Engineering, ICSE '15, pp. 156–166. IEEE Computer Society (2015)
13. Chen, L., Avizienis, A.: N-version programming: A fault-tolerance approach to reliability of software operation. In: International Symposium on Fault-Tolerant Computing, FTCS '78, pp. 113–119 (1978)
14. Chen, T.Y., Cheung, S.C., Yiu, S.M.: Metamorphic testing: a new approach for generating next test cases. Tech. rep., Department of Computer Science, Hong Kong University of Science and Technology (1998)

15. Cox, B., Evans, D., Filipi, A., Rowanhill, J., Hu, W., Davidson, J., Knight, J., Nguyen-Tuong, A., Hiser, J.: N-variant systems: A secretless framework for security through diversity. In: Proceedings of the Conference on USENIX Security Symposium, SEC '06. USENIX Association (2006)
16. Doong, R.K., Frankl, P.G.: The ASTOOT approach to testing object-oriented programs. ACM Transactions on Software Engineering and Methodology **3**(2), 101–130 (1994)
17. Fraser, G., Arcuri, A.: Evosuite: Automatic test suite generation for object-oriented software. In: Proceedings of the European Software Engineering Conference held jointly with the ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE '11, pp. 416–419. ACM (2011)
18. Gabel, M., Su, Z.: A study of the uniqueness of source code. In: Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE '10, pp. 147–156. ACM (2010)
19. Ghosn, M., Moses, F.: NCHRP report 406: Redundancy in highway bridge superstructures. Tech. rep., National Cooperative Highway Research Program (NCHRP), Transportation Research Board (1998). Http://onlinepubs.trb.org/onlinepubs/nchrp/nchrp_rpt_406.pdf
20. Ghosn, M., Yang, J.: NCHRP report 776: Bridge system safety and redundancy. Tech. rep., National Cooperative Highway Research Program (NCHRP), Transportation Research Board (2014). Http://onlinepubs.trb.org/onlinepubs/nchrp/nchrp_rpt_776.pdf
21. Goffi, A.: Automatic generation of cost-effective test oracles. In: ICSE '14: Proceedings of the 36th International Conference on Software Engineering, pp. 678–681. ACM (2014)
22. Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., Tonella, P.: Search-based synthesis of equivalent method sequences. In: Proceedings of the ACM SIGSOFT International Symposium on the Foundations of Software Engineering, FSE '14, pp. 366–376. ACM (2014)
23. Hamming, R.W.: Error detecting and error correcting codes. Bell System Technical Journal **29**(2), 147–160 (1950)
24. Harman, M., Langdon, W.B., Jia, Y., White, D.R., Arcuri, A., Clark, J.A.: The gismoe challenge: Constructing the pareto program surface using genetic programming to find better programs (keynote paper). In: Proceedings of the International Conference on Automated Software Engineering, ASE '12, pp. 1–14. ACM (2012)
25. Hindle, A., Barr, E.T., Su, Z., Gabel, M., Devanbu, P.: On the naturalness of software. In: Proceedings of the International Conference on Software Engineering, ICSE '12, pp. 837–847. ACM (2012)
26. Hosek, P., Cadar, C.: Varan the unbelievable: An efficient n-version execution framework. In: Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '15, pp. 339–353. ACM (2015)
27. Huang, W.C., Knottenbelt, W.J.: Self-adaptive containers: Building resource-efficient applications with low programmer overhead. In: Proceedings of the International Symposium on Software Engineering for Adaptive and Self-Managing Systems, SEAMS '13, pp. 123–132. IEEE Computer Society (2013)
28. Huang, Y., Kintala, C.M.R.: Software implemented fault tolerance technologies and experience. In: Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing, FTSC '93, pp. 2–9. IEEE Computer Society (1993)
29. Jiang, L., Su, Z.: Automatic mining of functionally equivalent code fragments via random testing. In: Proceedings of the International Symposium on Software Testing and Analysis, ISSTA '09, pp. 81–92. ACM (2009)
30. Kawrykow, D., Robillard, M.P.: Improving api usage through automatic detection of redundant code. In: Proceedings of the International Conference on Automated Software Engineering, ASE '09, pp. 111–122. IEEE Computer Society (2009)
31. Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**, 54–72 (2012)
32. Liu, W.D., Ghosn, M., Moses, F.: NCHRP report 458: Redundancy in highway bridge substructures. Tech. rep., National Cooperative Highway Research Program (NCHRP), Transportation Research Board (2001). Http://onlinepubs.trb.org/onlinepubs/nchrp/nchrp_rpt_458-a.pdf

33. Mattavelli, A.: Software redundancy: What, where, how. Ph.D. thesis, Università della Svizzera italiana (USI) (2016)
34. Mattavelli, A., Goffi, A., Gorla, A.: Synthesis of equivalent method calls in Guava. In: Proceedings of the 7th International Symposium on Search-Based Software Engineering, SSBSE '15, pp. 248–254. Springer (2015)
35. Misailovic, S., Sidiroglou, S., Hoffmann, H., Rinard, M.: Quality of service profiling. In: Proceedings of the International Conference on Software Engineering, ICSE '10, pp. 25–34. ACM (2010)
36. Patterson, D.A., Gibson, G., Katz, R.H.: A case for redundant arrays of inexpensive disks (RAID). SIGMOD Record **17**(3), 109–116 (1988)
37. Randell, B.: System structure for software fault tolerance. SIGPLAN Notes **10**(6), 437–449 (1975)
38. Reed, I.S., Solomon, G.: Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics **8**(2), 300–304 (1960)
39. Sadjadi, S.M., McKinley, P.K.: Using transparent shaping and Web services to support self-management of composite systems. In: Proceedings of the International Conference on Autonomic Computing, ICAC '05, pp. 76–87. IEEE Computer Society (2005)
40. Salamat, B., Jackson, T., Gal, A., Franz, M.: Orchestra: Intrusion detection using parallel execution and monitoring of program variants in user-space. In: Proceedings of the ACM SIGOPS EuroSys European Conference on Computer Systems, EuroSys '09, pp. 33–46. ACM (2009)
41. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: Proceedings of the 2010 IEEE Symposium on Mass Storage Systems and Technologies, MSST '10, pp. 1–10. IEEE Computer Society (2010)
42. Sidiroglou-Douskos, S., Lahtinen, E., Long, F., Rinard, M.: Automatic error elimination by horizontal code transfer across multiple applications. In: Proceedings of the Conference on Programming Language Design and Implementation, PLDI '15, pp. 43–54. ACM (2015)
43. Somani, A.K., Vaidya, N.H.: Understanding fault tolerance and reliability. IEEE Computer **30**(4), 45–50 (1997)
44. Subramanian, S., Thiran, P., Narendra, N.C., Mostefaoui, G.K., Maamar, Z.: On the enhancement of BPEL engines for self-healing composite web services. In: Proceedings of the International Symposium on Applications and the Internet, SAINT '08, pp. 33–39. IEEE Computer Society (2008)
45. Taher, Y., Benslimane, D., Fauvet, M.C., Maamar, Z.: Towards an approach for Web services substitution. In: Proceedings of the International Database Engineering and Applications Symposium, IDEAS '06, pp. 166–173. IEEE Computer Society (2006)
46. Trachsel, O., Gross, T.R.: Variant-based competitive parallel execution of sequential programs. In: Proceedings of the ACM International Conference on Computing Frontiers, CF '10, pp. 197–206. ACM (2010)
47. Weyuker, E.J.: On testing non-testable programs. The Computer Journal **25**(4), 465–470 (1982)
48. Xue, H., Dautenhahn, N., King, S.T.: Using replicated execution for a more secure and reliable web browser. In: Proceedings of the Annual Network and Distributed System Security Symposium, NDSS '12. The Internet Society (2012)
49. Yeh, Y.C.: Triple-triple redundant 777 primary flight computer. In: Proceedings of the IEEE Aerospace Applications Conference, vol. 1, pp. 293–307 (1996)