Software Qual J (2011) 19:753–769
DOI 10.1007/s11219-011-9151-x

Automating unit and integration testing with partial oracles

René Just · Franz Schweiggert

Published online: 12 July 2011

Abstract The oracle problem is an essential part in current research on automating software tests. Partial oracles seem to be a viable solution, but their suitability for different testing steps and general applicability for various systems remains still to be shown. This paper presents a study in which partial oracles are applied in order to automatically test a jpeg2000 encoder as an example for a modular software system with several integrated units and components. The effectiveness of the partial oracles is measured by means of mutation analysis to determine their adequacy for both unit and integration testing. Additionally, the paper presents possibilities of improving the effectiveness as well as the efficiency of the employed partial oracles. It shows how the knowledge of certain characteristics of the system to be tested, such as linearity or time-invariance, may lead to a better choice of partial oracles and thus to an improved effectiveness and efficiency.

Keywords Test automation · Partial oracles · Metamorphic testing · Integration testing · Mutation analysis · Random testing

1 Introduction

Increasing the level of automation is a crucial part in current research on software engineering and especially software testing. Automating software tests is a complex task which concerns not only the execution of test cases but also the generation of appropriate input values and the evaluation of the corresponding outputs. Selecting a suitable model for input value generation and an appropriate oracle to verify the outputs can be referred to as a *testing*

A preliminary version of this paper was presented at the 5th International Workshop on Automation of Software Test (AST'10)(Just and Schweiggert 2010)

R. Just (✉) · F. Schweiggert
Department of Applied Information Processing, Ulm University, 89069 Ulm, Germany
e-mail: rene.just@uni-ulm.de

F. Schweiggert
e-mail: franz.schweiggert@uni-ulm.de

strategy. In order to achieve reliable results from testing, this strategy has to cover the semantics of the implementation which is to be investigated since appropriate oracles and input values are correlated with both structure and processing logic.

Regarding the inputs, random or adaptive random generation is an established approach since it is suitable for most environments and an unbiased technique. The resulting outputs are evaluated by means of an *oracle* (Binder 1999). However, the usage of randomly generated inputs often results in the *oracle problem* (Weyuker 1982) if an appropriate oracle is not available. Regarding the oracle problem, various standard solutions (cf. Binder 1999) exist which are, however, only employable in rare situations. On the other hand, a promising class of oracles, the *partial oracles* (Weyuker 1982), are considered to be easily automatable, more often applicable and should therefore be used for automating software tests (Bertolino 2007). These oracles are referred to as partial oracles because they cannot determine the correct output of a system. They only exploit constraints of the underlying function or algorithm in order to identify faults within the tested system. A trivial example for a partial oracle concerning the trigonometric sine function is for instance the equation:

$$\sin(-x) = -\sin(x)$$

If this equation is not fulfilled by an implementation of the sine function, it can be judged to be faulty even without the knowledge of the correct output of $\sin(x)$ or $\sin(-x)$. However, an implementation which does not violate this constraint could still be faulty if it computes the same wrong output for $\sin(x)$ and $\sin(-x)$.

With regard to integration testing, where several units are combined to a subsystem, partial oracles have to be able to verify necessary conditions of the complete subsystem in order to be applicable. In comparison with unit testing, these necessary conditions of the complete subsystem might be less restrictive than the conditions of the individual units since they must hold for the complete subsystem. If, for instance, a software unit implements a linear time-invariant system, we can exploit the linearity to define necessary conditions that have to be fulfilled by the implementation. Now, if a subsystem that contains this unit is no longer linear and time-invariant, these necessary conditions do not hold for the integrated system and cannot be used as partial oracles. As a consequence, partial oracles applicable for the complete (sub)system could be less effective than partial oracles constructed for an individual unit of this system. Therefore, the question arises whether partial oracles are in principle adequate in the field of integration testing and, if so, how suitable they are.

This paper presents a study of constructing and applying partial oracles in order to automatically test several parts of an image processing application, more precisely a modular and object oriented jpeg2000 encoder. The study relies on random input generation and furthermore on mutation analysis to assess the effectiveness of the generated inputs as well as the applied partial oracles. It analyzes the adequacy of the chosen partial oracles for unit and integration testing and investigates additionally the complexity of the partial oracles as well as possible improvements of their effectiveness and efficiency. The remainder of the present paper is structured as follows: Sect. 2 deals with the basics of random input generation, partial oracles, and mutation analysis. The applied approach to evaluate the input values and partial oracles is also explained in this section. Thereafter, Sect. 3 describes the study in detail and discusses the corresponding results. Potential threats to validity are described in Sect. 4, and finally, Sect. 5 concludes the paper and discusses future work.

2 Preliminaries and related work

Concerning the input values, which have to be complete images in our case, it is obviously infeasible to cover all possibilities of the whole input range. Moreover, creating input values manually is time-consuming, particularly for image processing applications which deal with complex input values, and hence not convenient. Thus, we rely on random input generation in order to create the inputs efficiently since this technique is simple and versatilely applicable in most environments. Various models for generating gray-level images, such as the random or boolean model, are available, and in our study, the random model is applied, which determines the gray scale value of each pixel independently.

In order to obtain color images, the random model for gray-level images has been extended as shown in Fig. 1. Each color component of the RGB color space is created independently by means of the random model and the complete image is achieved by merging all three components. That means that a gray-scale image is randomly generated for every color component red (R), green (G), and blue (B). Then, these images are interpreted as color components, i.e., their gray-scale values represent the according color values of red, green, or blue. Finally, the union of the color components forms the resulting, randomly generated, color image. Generating a gray-scale image randomly is equal to computing a matrix in which every coefficient, which represents a pixel, is randomly generated. Since the resulting matrix shall represent a gray-scale image, the values of all coefficients have to be restricted to the interval [0,255]

$$\Longleftrightarrow \begin{bmatrix} 128 & 128 & 128 & 128 & 128 & 128 \\ 128 & 128 & 128 & 128 & 128 & 128 \\ 70 & 70 & 70 & 70 & 70 & 70 \\ 70 & 70 & 70 & 70 & 70 & 70 \\ 170 & 170 & 170 & 170 & 170 & 170 \\ 170 & 170 & 170 & 170 & 170 & 170 \end{bmatrix}$$

With regard to the randomly generated input values, we face the oracle problem, which is in our case to be alleviated by means of partial oracles. However, this class of oracles can only check necessary conditions but cannot verify sufficient conditions. Thus, one of the main characteristics of partial oracles is that their results may be false negative, i.e., if the oracle judges the System Under Test (SUT) to be correct, it may still contain a fault

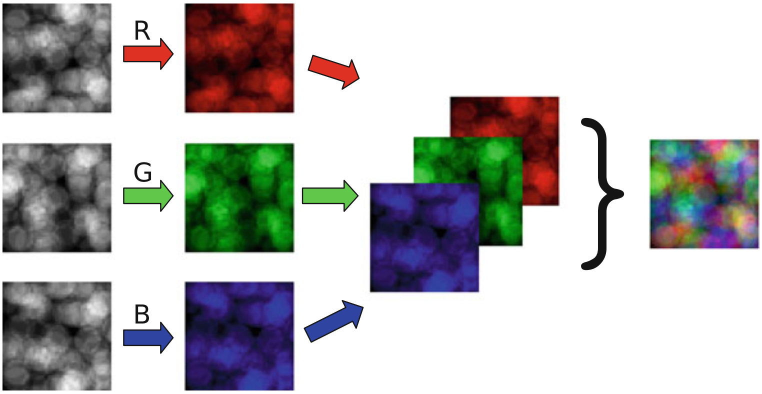


Fig. 1 Input generation model for color images

since it only fulfills the necessary conditions. On the other hand, if such an oracle reveals a defect in the SUT, the system contains definitely a fault. It thus becomes clear that assessing the oracles is necessary since they may not detect every fault and the effectiveness of the applied testing strategy depends on the quality of the oracle.

Associated with the class of partial oracles is for instance metamorphic testing which exploits properties of the SUT in order to evaluate the corresponding dependencies between inputs and outputs. The properties are described using metamorphic relations (cf. Chen et al. 2003) consisting of two relations R_I and R_O. Assume I to be the input domain, O the output range, and f a mapping $f: I \rightarrow O$. In addition, $R_I \subseteq I^n$ and $R_O \subseteq I^n \times O^n$. The pair (R_I, R_O) is called a metamorphic relation if and only if the following implication is fulfilled:

$$(i_1, \ldots, i_n) \in R_I \Rightarrow (i_1, \ldots, i_n, f(i_1), \ldots, f(i_n)) \in R_O$$

An example for testing with such partial oracles is depicted in Fig. 2 where the relation between the inputs R_I is the matrix transposition. The relation between the resulting outputs R_O is also the matrix transposition because of the commutativity of the two-dimensional wavelet transformation, which is the SUT in this case. The workflow of metamorphic testing can thus be described briefly as given below (c.f. Zhou et al. 2004).

1. Generate a follow-up test case from an arbitrary input according to a relation R_I.
2. Execute the SUT independently with both inputs.
3. Verify whether the resulting outputs fulfill the corresponding relation R_O.

As already mentioned, the applied partial oracles have to be assessed in order to achieve reliable test results. For this purpose, mutation analysis is used which is a fault-based approach, originally introduced in (Budd 1980, DeMillo et al. 1978), and appropriate for testing and benchmarking purposes (Andrews et al. 2005, Guderlei et al. 2008). Aiming at measuring the quality of a given testing strategy, faults are seeded

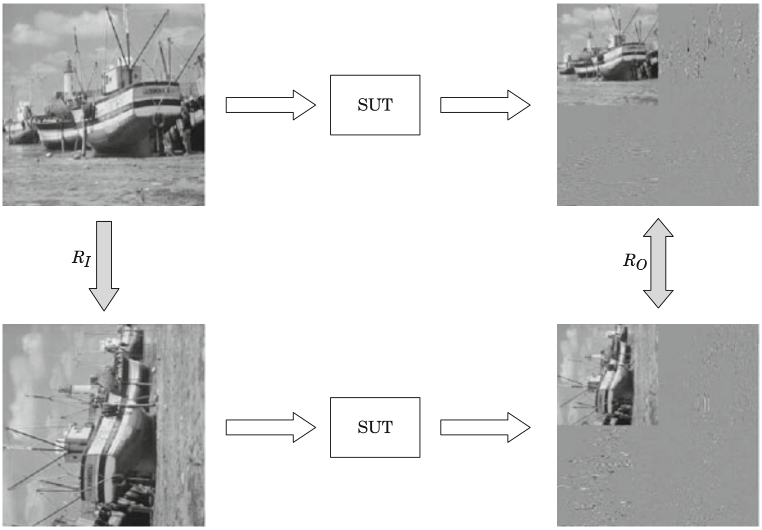


Fig. 2 Exploiting the commutativity of the two-dimensional wavelet transformation as partial oracle by means of the matrix transposition

systematically into the SUT and the corresponding testing strategies are verified with regard to their ability to reveal the injected faults. The way of applying mutation analysis is formally specified by *mutation operators* (Ma et al. 2002, Offutt et al. 1996), and the resulting faulty versions of the SUT are referred to as *mutants*. The mutation operators can be distinguished between traditional and class-based operators (cf. Ma et al. 2005). Traditional operators are applied at the functional or method level and the class-based ones operate at class level in object-oriented systems. If a fault is revealed by a test case, the corresponding mutant is said to be *killed*. Accordingly, relating the number of all killed mutants to the total number of generated mutants is appropriate to measure the fault-finding capability of the applied testing strategy. However, a mutant cannot be killed under all circumstances. In fact, when the mutation does not affect the semantics of the SUT, there exists no test case that can detect the mutation. In this case, the mutant is said to be *equivalent* (c.f. Offutt and Pan 1997). Thus, disassociating equivalent mutants from the complete repertoire provides an improved assessment of the effectiveness with the subsequent measure, the *mutation score MS*:

$$MS = \frac{M_k(Number\ of\ killed\ mutants)}{M_t(Number\ of\ non\text{-}equivalent\ mutants)}$$

In order to kill a mutant, three necessary conditions have to be fulfilled (cf. Voas 1992):

1. The mutated code has to be reached and executed.
2. The mutation has to change the state of the program.
3. The change has to be propagated to the output.

Obviously, the first condition (reachability) is solely related to the input values, apart from dead code fragments, and the latter can be reduced to the question of semantic equivalence. Since a testing strategy consists of an input generation model and an oracle to evaluate the outputs, the adequacy of a strategy depends on both the quality of the input values and the capability of the oracle. However, the effectiveness of the oracle is correlated with the quality of the input values according to the reachability condition.

Mutation analysis can be employed to assess both parts of the strategy. First, the input values can be evaluated with the original implementation as a perfect oracle. The resulting mutation score MS_I for the inputs provides an upper bound for the mutation score of the complete strategy since the perfect oracle, i.e., the best available oracle, is applied in this step. The (partial) oracle of the strategy can then be assessed with the input values that have been determined in the first step. Only the mutants killed by the perfect oracle are used in the second step because all other mutants cannot be killed. It has to be pointed out that the perfect oracle is never applied in this second step because a mutant is said to be killed, by the oracle which is to be assessed, if and only if it violates the constraints represented by the applied (partial) oracle. Hence, the mutation score MS_O of the investigated oracle represents the number of mutants killed by this oracle related to the number of killable mutants (i.e., mutants killed by the perfect oracle). Now, in order to express the dependency of MS_O and MS_I, we have to define the overall mutation score MS_S for a complete testing strategy as:

$$MS_S = MS_I \cdot MS_O$$

Let M_k^p be the number of mutants killed by the perfect oracle with respect to the applied input generation model. Furthermore, M_t denotes again the total number of non-equivalent mutants of the SUT, and M_k represents the number of mutants killed by the employed (partial) oracle. The mutation score MS_S is equal to MS:

$$MS_S = MS_I \cdot MS_O = \frac{M_k^p}{M_t} \cdot \frac{M_k}{M_k^p} = \frac{M_k}{M_t} = MS$$

The separated evaluation, however, provides a better view on the effectiveness of the individual parts of the testing strategy. Let us consider the following example:

- Number of non-equivalent mutants: 900
- Mutants killed by the perfect oracle: 882
- Mutants killed by the (partial) oracle: 750

By employing the above equation, we obtain the corresponding mutation scores:

$$MS_S = \frac{882}{900} \cdot \frac{750}{882} = 0.98 \cdot 0.85 = 0.83 = \frac{750}{900} = MS$$

In order to increase the effectiveness of the assessed testing strategy, it would be advisable in this case to focus on the applied oracle because the input generation model yields a satisfying result of 0.98. Aiming at evaluating the applied oracle exclusively, reliable results can only be achieved for almost perfect input values, i.e., $MS_I \approx 1$.

The effectiveness of the generated inputs might be assessed with other metrics. The degree of certain code coverage criteria such as statement or branch coverage is, for example, also applicable for this purpose (cf. Zhu et al. 1997). Nevertheless, we rely exclusively on mutation analysis in this study. It has to be pointed out that mutation analysis is not feasible without appropriate tool support. Tools are available for lots of common programming languages like Fortran, C#, or PHP. MuJava (Ma et al. 2005, MuJava. 2009) is for instance an established tool in the field of Java applications and used in the present paper.

3 Case study

The selected SUT is a Java implementation of the jpeg2000 (2010) encoder which is part of the JJ2000 (2010) library. This encoder is a system consisting of several concatenated subsystems that in turn consist of various combined transformations. The system to be investigated, illustrated in Fig. 3, is a complex and large subsystem which is responsible for preprocessing and decorrelation. The size, measured in Lines of Code (LOC), of the individual parts and the complete subsystem is depicted in Table 1. The overall size of the jpeg2000 encoder and the jj2000 library is 14k LOC and 30k LOC, respectively.

Generally, the selected SUT takes, as shown in Fig. 3, an uncompressed color image as an input value and executes the following workflow:

1. Split color image into color components red (R), green (G), and blue (B).
2. Shift the color values of each color component.
3. Transform RGB color components into YCbCr components.
4. Decompose components by applying the two-dimensional wavelet transformation.

As a consequence of this workflow, the outputs of the SUT are three individually decomposed color components where each of them contains a DC component (approximation of the color values) and three detail components (differences between pixels).

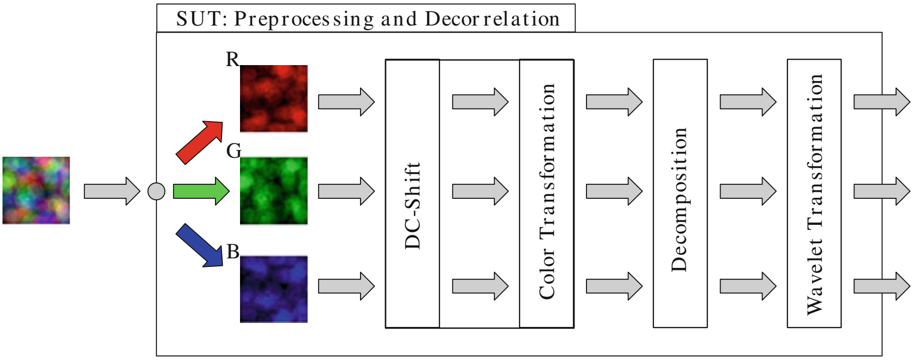


Fig. 3 The investigated subsystem for preprocessing and decorrelation

Table 1 Software packages and physical lines of code of the investigated subsystem

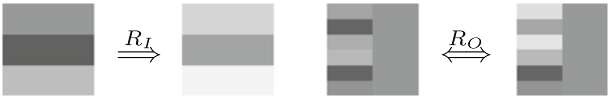
	DC-shift/color transformation	Decomposition	Wavelet transformation	Complete subsystem
LOC*	1422	964	2010	4396
Package	jj2000.j2k.image	jj2000.j2k.wavelet	jj2000.j2k.wavelet	jj2000.j2k

* LOC as reported by sloccount (non-comment and non-blank lines)

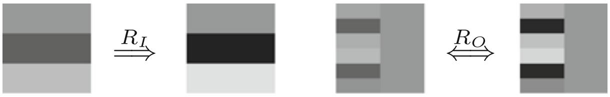
Further background information on the encoder as well as the complete jpeg2000 standard is described in detail e.g., in (Christopoulos et al. 2000, Skodras et al. 2001).

Generally, the usage of randomly generated inputs, as applied in this study, results in the oracle problem for image processing applications. Therefore, handcrafted or well-known standard test images are usually employed for which the expected output can be defined in advance. However, the oracle problem is avoided in our study, as already mentioned in Sect. 1, by means of partial oracles. Aiming at automatically testing this integrated subsystem with partial oracles, the oracles have to be applicable to the complete subsystem. Therefore, the following oracles have been chosen:

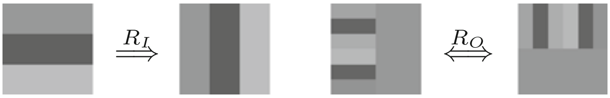
R1: A constant offset is added to every color value of the input image. Because of this constant offset, the mean color value of each color component changes but the difference between two pixels remains the same. Therefore, only the DC component must be increased (or decreased if the offset is a negative value) by the constant offset.



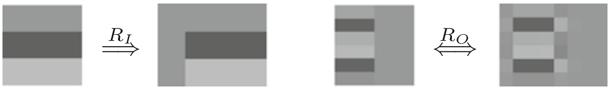
R2: The color values of the input image are multiplied by a constant factor. As a consequence, the mean color value as well as the differences between pixels are affected. Thus, the DC component and all detail components have to be changed with respect to the constant factor.



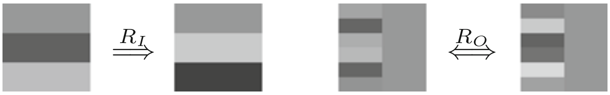
R3: The input image is transposed by means of the standard matrix transposition. Because of the linearity of the SUT and the commutativity of the two-dimensional wavelet transformation, the resulting components (DC component and detail components) have to be transposed as well.



R4: The pixel values of each row within the input image can be regarded as a signal, and since the SUT is linear and time-invariant, these signals can be shifted. For this purpose, the image width is enlarged with a defined number (constant for all rows) of leading zeros. Consequently, the resulting components also have to be shifted.



R5: Applying an inverted image to the SUT has to result in inverted components due to the linearity of the SUT. Hence, the color values of the input image are inverted, and all resulting components have to be affected.



All oracles can be described as metamorphic relations and summarized as follows:

R1 R_I: Add an offset to the color values.
R_O: Only the DC component must be affected.
R2 R_I: Multiply the color values by a coefficient.
R_O: Every pixel has to be affected.
R3 R_I: Transpose the pixel array of the input image.
R_O: The resulting components have to be transposed.
R4 R_I: Enlarge the input image with zero-padding.
R_O: The resulting components have to be shifted.
R5 R_I: Invert the color values of the input image.
R_O: The color values of the resulting components have to be inverted.

The metamorphic relations are implemented as matrix transformations. This means that on the one hand the color components of the randomly generated input values are mapped to follow-up matrices (according to R_I). On the other hand, the resulting outputs of the execution of the follow-up test cases are normalized (i.e., they are transformed according to the corresponding relation R_O), and then, they are compared with the output of the randomly generated input.

Based on the bipartite approach already mentioned in Sect. 2, we investigate the adequacy of these partial oracles for testing purposes with respect to the integrated subsystem. In total, 1977 non-equivalent traditional mutants can be generated for the complete subsystem. Additionally, 206 non-equivalent class-based mutants can be obtained (e.g, in interfaces between the transformations). The following examples of mutation operators illustrate the difference between traditional and class-based mutants:

Traditional mutants

```
– int a = b + c;      ⟹  int a = b - c;
– if(a && b){...};    ⟹  if(a || b){...};
```

Class-based mutants

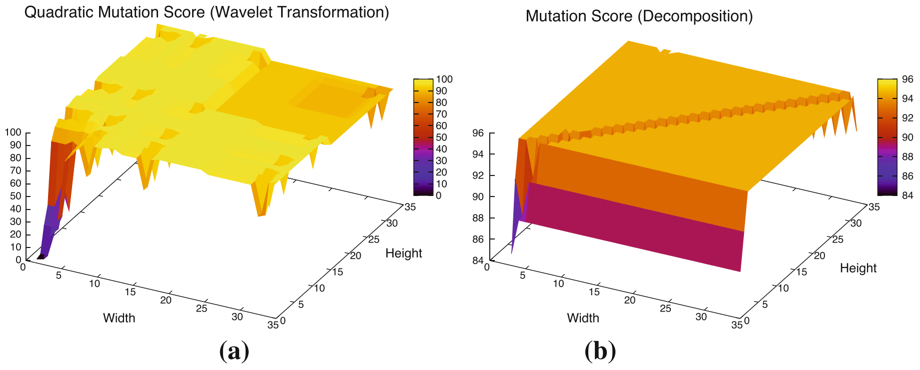
```
– component.setWidth(5);  ⟹  component.setHeight(5);
– A obj = new A1();       ⟹  A obj = new A2();
```

The class-based mutants represent structural defects in contrast to the traditional mutants which are injected at the functional level. Thus, the class-based mutants can be regarded as faults that could have been introduced by a programmer during the integration of software units. It has to be pointed out that the class-based mutants are obligatory in this study since we aim at assessing the adequacy of partial oracles for integration testing.

3.1 Evaluation of input values

Corresponding to the reachability condition, the mutated code has to be covered in order to be detected. Hence, the effectiveness of the partial oracles is heavily dependent on the input values. Consequently, we assess the input values in a first step, as described in Sect. 2, to provide the most adequate inputs for evaluating the partial oracles. Considering the input values, different properties like the image dimension (i.e., width and height) or the color depth may affect the suitability. However, it turned out that only the image dimension has an impact. Thus, images with different width and height are applied to determine the most appropriate inputs.

In order to achieve the highest possible mutation score for the input values MS_I, we use an exhaustive search over a limited and reduced search space. Therefore, we apply randomly generated images, employ the original implementation as a *Golden Standard Oracle* (Binder 1999) (i.e., as the perfect oracle), and use the mutation score as fitness function. The remaining mutants not killed after the search are inspected to reject the equivalent mutants and to get the correct mutation score. This task is done manually but approaches exist to identify several equivalent mutants automatically (cf. Offutt and Pan



(a)

(b)

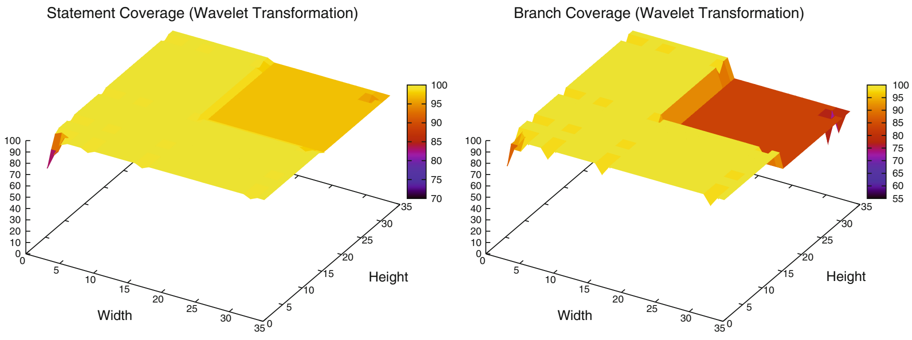
Fig. 4 Fitness landscapes for **a** the wavelet transformation and **b** the decomposition

1997). Two examples for resulting fitness landscapes are shown in Fig. 4a, b. The input values are classified according to their effectiveness (mutation score). That means, all images within a certain equivalence class yield exactly the same mutation score when employing them as input value and applying the perfect oracle. More precisely, the same mutants are killed by all images of the same equivalence class. Considering, for example, the fitness landscape of the Decomposition in Fig. 4b, we could identify exactly three equivalence classes in this case:

1. $width < height$
2. $width = height$
3. $width > height$

However, no class achieves a mutation score of 100%, this means that several runs of the SUT are necessary to collectively reach the full mutation score. In general, if the input images related to the most effective class are sufficient to kill all non-equivalent mutants, a conjunction of multiple classes, which implies several runs of the SUT, is not necessary. Otherwise, the classes have to be combined to collectively kill all of the mutants. As one can see by means of Fig. 4a, multiple executions are not necessary in this case since input images exist which achieve a mutation score of 100%.

As already mentioned in Sect. 2, another metric could be applied as fitness function in this first step. For this reason, we investigate two code coverage metrics as fitness function in addition to the mutation analysis. We use statement and simple branch coverage within the same search space. An advantage of both code coverage metrics is that of their being less expensive than mutation analysis. For every input value, the SUT has to be evaluated just once. However, it turns out that both metrics are weaker criteria than mutation analysis. Considering the three conditions to kill a mutant, the first one (reachability) is obviously a necessary condition for all three metrics. In contrast to the mutation analysis, this condition is, however, also sufficient for the code coverage metrics. Since we apply all mutation operators, available within the mutation tool, every basic block and expression is mutated. In addition, almost every statement is mutated. Thus, mutation analysis implies both code coverage criteria in this case and is according to this fact the stronger criterion. Example results of the exhaustive searches are depicted in Fig. 5a, b. As one can see, the coverage criteria achieve quite often a degree of 100%. Compared with the mutation score of the diagram in Fig. 4a, it becomes clear that input values exist which cover all statements and branches, but they are not able to kill all of the mutants.



(a) Statement coverage

(b) Branch coverage

Fig. 5 Code coverage for the wavelet transformation as reported by cobertura

It has to be pointed out that search-based techniques (McMinn 2004) may be more efficient for larger search spaces. Additionally, the exhaustive approach is no longer feasible for huge input domains. Since we exactly know where the mutation is located in the source code, one could alternatively transform the problem of searching adequate inputs, which cover the mutation, into a path problem (c.f. Visser et al. 2004).

3.2 Evaluation of the partial oracles

The partial oracles are investigated with regards to both the capability to reveal faults in each transformation and the overall effectiveness for the complete subsystem. It has to be pointed out that roughly 40% of the traditional and 20% of the class-based mutants throw a runtime exception, when the corresponding mutated code is covered, mostly because of invalid array indexes or references. Due to the restrictions of the Java programming language, these faults could thus be revealed without a particular oracle. However, all partial oracles catch these exceptions and mark the corresponding mutants as killed in this study. Consequently, the number of detected faults includes those mutants killed by an exception.

The effectiveness of the partial oracles varies notably with respect to a specific transformation, as illustrated in Table 2. Concerning the DC-shift in conjunction with the color transformation, the relation `R1`, which kills 504 mutants, is for instance much more effective than `R4`, which reveals only 394 defects. In reference to the total number of 608 non-equivalent mutants, this represents a considerable discrepancy in the mutation score between 83 and 65%.

In addition, the effectiveness of the applied oracles differs with regard to the specific parts of the subsystem, and thus, none of them is adequate for all transformations. Considering, for example, the DC-shift and color transformation, the relation `R1` yields the highest mutation score for this part of the subsystem. However, in the field of the wavelet transformation, the effectiveness of `R1` is only average in comparison with the other relations.

Furthermore, there are substantial differences in the effectiveness for the different kind of mutants as shown in Table 3. The relation `R3` is for instance the most effective oracle concerning the traditional mutants with an overall mutation score of 91% and ranging between 80 and 97% with regard to the individual transformations. It is, however, rather poor in killing the class-based mutants with a ratio of approximately 58%. Hence, it would be a fallacy to conclude that an oracle that is highly effective for testing a specific transformation,

Table 2 Effectiveness of the applied partial oracles for the particular transformations and the complete subsystem concerning the traditional mutants

	DC-shift/color transformation	Decomposition	Wavelet transformation	Complete subsystem
Total	608	441	928	1977
R1	504 (82.89%)	402 (91.16%)	860 (92.67%)	1766 (89.33%)
R2	489 (80.43%)	381 (86.39%)	859 (92.56%)	1729 (87.46%)
R3	482 (79.28%)	413 (93.65%)	904 (97.41%)	1799 (91.00%)
R4	394 (64.80%)	398 (90.25%)	876 (94.40%)	1668 (84.37%)
R5	456 (75.00%)	372 (84.35%)	781 (84.16%)	1609 (81.39%)

Table 3 Comparison of the effectiveness of the applied partial oracles for the complete subsystem with respect to the different kind of mutants

	Traditional mutants	Class-based mutants
Total	1977	206
R1	1766 (89.33%)	151 (73.30%)
R2	1729 (87.46%)	146 (70.87%)
R3	1799 (91.00%)	119 (57.77%)
R4	1668 (84.37%)	84 (40.78%)
R5	1609 (81.39%)	87 (42.23%)

or more generally an individual unit of a subsystem, is as a consequence of this equally suitable for integration testing. Moreover, the effectiveness of the applied relations is insufficient concerning the class-based mutants, even though the results achieved by the relations are predominantly sufficient with respect to the traditional mutants.

In order to increase the effectiveness and achieve satisfying results, the partial oracles may be combined. Therefore, the two most effective oracles are combined pairwise, and additionally, the overall effectiveness of the partial oracles altogether is investigated. A combination in this case means that all of the combined partial oracles are applied individually, and the mutation score collectively achieved equates to the effectiveness of the combined oracles. The results for each transformation as well as for the complete subsystem are depicted in Table 4, where Rx and Ry represent the most effective and the second most effective oracle with respect to the corresponding transformation. Regarding, for instance, the wavelet transformation, R3 is the most effective oracle and R4 is the second most effective one. Thus, the column ”Rx & Ry“ denotes the mutation score collectively achieved by the combination of these both oracles.

As one can see, the combination of the partial oracles can significantly increase their effectiveness, especially with regard to the class-based mutants. The conjunction of the two most effective relations leads to a mutation score of 94% for the class-based mutants and an overall mutation score of 95% for all traditional mutants. In addition, the variance is reduced significantly since all ratios are not less than 90%.

On the other hand, the additional benefit of combining all partial oracles is rather small compared with the pairwise combination of the two most effective ones. The necessary

Table 4 Increase in effectiveness of the applied partial oracles by means of combination of the two most effective oracles (Rx & Ry) and all oracles (R1, R2, R3, R4, and R5)

	Total	Rx	Ry	Rx & Ry	All oracles
Traditional mutants					
DC-shift/color transformation	608	R1	R2	544 (89.47%)	553 (90.95%)
Decomposition	441	R3	R1	422 (95.69%)	430 (97.51%)
Wavelet transformation	928	R3	R4	923 (99.46%)	923 (99.46%)
Complete subsystem	1977	R3	R1	1889 (95.55%)	1906 (96.41%)
Class-based mutants					
Complete subsystem	206	R1	R2	193 (93.69%)	200 (97.09%)

effort of applying two more oracles is disproportionate compared with the further increase of at most 2% for the traditional and 3% for the class-based mutants.

3.3 Efficiency and effectiveness improvements

Considering the complexity of the partial oracles, we can distinguish between code and model complexity. Metrics for the code complexity are for instance lines of code (LOC) or McCabe's cyclomatic complexity which is a measure for the structural complexity. Based on the control flow graph, let N be the number of nodes, E the number of edges, and P the number of connected components. McCabe's cyclomatic complexity C is then defined as:

$$C = E - N + 2 \cdot P$$

Both metrics are indicators for the effort to implement the corresponding partial oracle.

On the other hand, we use the term model complexity for the costs of applying the partial oracle. The model complexity is predominantly defined by the number of parameters and additionally by the number of necessary inputs, which is equal to the number of required runs of the SUT. Obviously, the number of parameters is more severe since the partial oracle has to be applied for every parameter value which is to be investigated. Considering for example the oracle R2 which can be described as:

$$\underbrace{SUT(c \cdot I)}_{run\#1} = c \cdot \underbrace{SUT(I)}_{run\#2}$$

The only parameter of this oracle is the factor c which has to be chosen. In addition, the SUT has to be executed twice in order to apply this oracle. Thus, the partial oracle expects two inputs, namely I and $c \cdot I$. It has to be pointed that the difference between parameter an input is of particular importance. The oracle needs to be calibrated if it contains a parameter, and the choice of the parameter value has an impact on the effectiveness. The code and model complexity as well as the overall mutation score of the investigated oracles are depicted in detail in Table 5.

Given the complexity and effectiveness (mutation score) of the employed partial oracles, we focus on the oracles R1 and R3 for further improvements of effectiveness and efficiency since they achieve the highest mutation score for the class-based and traditional mutants, respectively. In order to avoid the additional parameter of R1, the constant offset, we can generalize this partial oracle by adding a randomly generated offset to each coefficient. For this purpose, the new partial oracle R6 generates another random image R with the same dimension as the input I and adds both together with the standard matrix

Table 5 Complexity and effectiveness of applied partial oracles (Params denotes the number of parameters of the corresponding oracle and Inputs represents the number of necessary runs of the SUT)

	LOC	McCabe		Params	Inputs	Mutation score	
		R_I	R_O			Traditional	Class-based
R1	363	17	11	1	2	89.33%	73.30%
R2	352	17	6	1	2	87.46%	70.87%
R3	327	10	5	0	2	91.00%	57.77%
R4	398	20	11	1	2	84.37%	40.78%
R5	331	17	6	0	2	81.39%	42.23%

Table 6 Complexity and effectiveness of enhanced partial oracles (Params denotes the number of parameters of the corresponding oracle and Inputs represents the number of necessary runs of the SUT)

	LOC	McCabe		Params	Inputs	Mutation score	
		R_I	R_O			Traditional	Class-based
R6	368	18	9	0	3	93.17%	88.83%
R7	391	24	13	0	3	96.26%	96.60%

addition. Because of the linearity of the SUT, the necessary condition that has to be fulfilled by the SUT can be described by the following equation:

$$(\mathbf{R6})\quad \underbrace{SUT(I+R)}_{run\#1} = \underbrace{SUT(I)}_{run\#2} + \underbrace{SUT(R)}_{run\#3}$$

Therefore, the additional parameter has been replaced by another input and thus an extra run of the SUT. Moreover, the implementation of R_O for R6 is even simpler compared with R1 because we do not have to locate the DC component in the outputs.

According to the increase in effectiveness by combining partial oracles, a combination of the oracles R3 and R6 seems to be promising for further improvements. With respect to efficiency, especially execution time, we can again exploit the linearity and furthermore the commutativity of the SUT. To combine both necessary conditions of R3 and R6 within one oracle, we define a new partial oracle R7:

$$(\mathbf{R7})\quad \underbrace{SUT((I+R)^T)}_{run\#1} = (\underbrace{SUT(I)}_{run\#2} + \underbrace{SUT(R)}_{run\#3})^T$$

The model complexity of this partial oracle is equal to that of R6 because we do not have additional parameters and the required number of executions of the SUT is still 3. Hence, this oracle leads to an efficiency improvement by reducing the runtime by 40% in comparison with single executions of R3 and R6. The complexity and mutation score of the enhanced oracles are illustrated in Table 6.

As one can see, these oracles are more complex but they achieve a significant increase in effectiveness. Since the variance of the mutation score for the individual parts is also reduced considerably, between 91.44 and 98.81%, they can be regarded as suitable for the SUT.

3.4 Discussion

Regarding the results, it seems that partial oracles are indeed applicable for integration testing, but a few aspects have to be considered. First of all, the partial oracles derived from the characteristics of the integrated (sub)system may be less effective than partial oracles for the individual parts of this system. In addition, the effectiveness of oracles for testing the processing logic of system parts most likely differs from the effectiveness for testing the integration of these parts. In order to compensate such variations and to increase the effectiveness, it is advisable to combine the partial oracles.

As shown in the case study, combining the most effective oracles is nearly as powerful as joining all oracles. Thus, the testing effort can be reduced without major quality losses by prioritizing the partial oracles with regard to their fault-finding capability and joining just the most effective ones. Since the SUT has to be executed for every partial oracle and every input value, the time needed to run all tests is proportional to the number of partial oracles. Concerning the investigated subsystem, processing the input values is a time-consuming task and thus applying only two of four relations would halve the time totally needed. This decrease in the execution time, and hence testing effort, may be of particular importance in the field of regression testing.

According to our results of the efficiency and effectiveness improvements, some suggestions on the selection and construction of partial oracles can be given. First of all, it is advisable to exploit constraints like equivalence relations in conjunction with properties such as commutativity, distributivity, or associativity. For efficiency reasons, the combination of necessary conditions should be implemented within one partial oracle even though the complexity is increasing. Additional parameters should be avoided or kept to a minimum since they extend the search space. Moreover, with respect to automated (adaptive) random testing, a partial oracle like R7 would be preferable (e.g., via prioritization) because it generates a follow-up value with different properties (cf. Chen et al. 2004). As a consequence of this, the input values are better distributed in the search space and the convergence rate of the mutation score is most likely higher.

Since the investigated SUT is a Java implementation, there are many mutants that result in an exception due to violating restrictions. As mentioned above, these mutants can be killed without a specific oracle, e.g., with smoke tests which can be regarded as the simplest partial oracle. Thus, partial oracles can be implicitly combined with such smoke tests, for instance, by means of an adequate exception handling, or smoke tests can be used as a first step to reveal invalid indexes and references. However, this is language dependent and may be less suitable for other languages.

4 Threats to validity

With regard to the discussed results, some threats to validity have to be considered. The chosen mutation operators could be a threat to internal validity. Different operators or hand-seeded faults may affect the mutation score of the investigated partial oracles. However, we applied all possible operators, provided by the mutation tool, in order to cover a wide variety of defects. Furthermore, the applied operators are frequently used in the literature and therefore provide comparable results (Andrews et al. 2005).

A potential threat to external validity might be the representativeness of the selected application. There is no guarantee that the depicted results and the achieved improvements of effectiveness and efficiency of the partial oracles will be the same for other systems. However, the investigated subsystem represents a modular object oriented application with several integrated units and hence is comparable to other software systems. So, we judge that the reported results are meaningful. Nevertheless, a replication of this study is necessary, especially for other programming languages and different software systems. This matter is left open for future research.

Defects in the mutation tool or in our testing framework could be a threat to construct validity, but we controlled this threat by analyzing the generated mutants and by testing our implementation. Every partial oracle was applied to the original implementation and executed with all input values to ensure that the implemented constraints are fulfilled by the investigated system. Thus, we judge that the mutants were properly generated and that our implementation worked correctly.

5 Conclusions and future work

Applying partial oracles to integration testing has been addressed in this paper. In order to evaluate the applicability of partial oracles for this purpose, an integrated subsystem of an object oriented image processing application is investigated by means of mutation analysis. The applied partial oracles are assessed with regard to their capability to reveal faults in the individual parts of the subsystem and their suitability for integration testing. It turns out that the effectiveness of the investigated partial oracles varies concerning the different parts of the subsystem and none of the oracles is sufficient for the complete subsystem. Additionally, the adequacy of the partial oracles for integration testing cannot be inferred from the effectiveness for testing the particular parts of the subsystem. However, combining the partial oracles yields satisfying results for both unit testing and integration testing. Moreover, exploiting certain characteristics of the system under test provides partial oracles which lead to a significant increase in effectiveness and efficiency. Hence, this kind of oracle seems to be suitable for testing purposes and especially test automation with respect to the oracle problem.

In summary, it can be said that partial oracles are suitable for automating various parts of the software testing process but further research is necessary to confirm the results. In addition, our results should be transferable to other linear and time-invariant systems as well as transformations which meet the mentioned constraints. Examining the transferability to other partial oracles as well as different software systems will thus be part of our future work. Furthermore, conducting a study with similar applications written in different programming languages to compare the effectiveness of the partial oracles with respect to the corresponding language is another area for future work.

References

Andrews, J. H., Briand, L. C., & Labiche, Y. (2005). Is mutation an appropriate tool for testing experiments? In: *Proceedings of the 27th International Conference on Software Engineering, ICSE '05* (pp. 402–411).

Bertolino, A. (2007). Software testing research: Achievements, challenges, dreams. In: *2007 Future of Software Engineering, FOSE '07* (pp. 85–103).

Binder, R. (1999). *Testing object-oriented systems: Models, patterns, and tools*. Reading: Addison-Wesley.

Budd, T. A. (1980). Mutation analysis of program test data. Ph.D. thesis, Yale University.

Chen, T. Y., Huang, D. H., Tse, T. H., & Zhou, Z. Q. (2004). Case studies on the selection of useful relations in metamorphic testing. In: *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering, JIISIC '04* (pp. 569–583).

Chen, T. Y., Tse, T. H., & Zhou, Z. Q. (2003). Fault-based testing without the need of oracles. *Information and Software Technology, 45*(1), 1–9.

Christopoulos, C., Skodras, A., & Ebrahimi, T. (2000). The jpeg2000 still image coding system: An overview. *IEEE Transactions on Consumer Electronics, 46*(4), 1103–1127.

DeMillo, R. A., Lipton, R. J., Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *IEEE Computer, 11*(4), 34–41.

Guderlei, R., Just, R., Schneckenburger, C., & Schweiggert, F. (2008). Benchmarking testing strategies with tools from mutation analysis. In: *Proceedings of the 2008 IEEE International Conference on Software Testing Verification and Validation Workshop, ICSTW '08* (pp. 360–364).

(JJ2000. 2010) Official web site of the jj2000 project. http://jj2000.epfl.ch.

(Jpeg. 2010) Official web site of the joint photographic experts group. http://www.jpeg.org.

Just, R., Schweiggert, F. (2010). Automating software tests with partial oracles in integrated environments. In: *Proceedings of the 5th Workshop on Automation of Software Test, AST '10* (pp. 91–94).

Ma, Y. S., Kwon, Y. R., Offutt, J. (2002). Inter-class mutation operators for Java. In: *Proceedings of the 13th International Symposium on Software Reliability Engineering, ISSRE '02* (pp. 352–363).

Ma, Y. S., Offutt, J., & Kwon, Y. R. (2005). Mujava: An automated class mutation system. *Software Testing, Verification, and Reliability, 15*(2), 97–133.

McMinn, P. (2004). Search-based software test data generation: a survey. *Software Testing, Verification, and Reliability, 14*(2), 105–156.

MuJava. (2009). The web site of the MuJava project. http://cs.gmu.edu/offutt/mujava.

Offutt, A., & Pan, J. (1997). Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification, and Reliability, 7*(3), 165–192.

Offutt, A. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology, 5*(2), 99–118.

Skodras, A., Christopoulos, C., & Ebrahimi, T. (2001). The jpeg 2000 still image compression standard. *IEEE Signal Processing Magazine, 18*(5), 36–58.

Visser, W., Pasareanu, C. S., Khurshid, S. (2004). Test input generation with java PathFinder. In: *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA '04* (pp. 97–107).

Voas, J. M. (1992). PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering, 18*(8), 717–727.

Weyuker, E. J. (1982). On testing non-testable programs. *The Computer Journal, 25*(4), 465–470.

Zhou, Z. Q., Huang, D. H., Tse, T. H., Yang, Z., Huang, H., Chen, T. Y. (2004). Metamorphic testing and its applications. In: *Proceedings of the 8th International Symposium on Future Software Technology, ISFST '04*.

Zhu, H., Hall, P. A. V., & May, J. H. R. (1997). Software unit test coverage and adequacy. *ACM Computing Surveys, 29*(4), 366–427.

Author Biographies

René Just received his M.Sc. in Computer Science from Ulm University, Germany in 2008. Since 2008 he is a PhD student at Ulm University in the Institute of Applied Information Processing. His research interests include software test automation with respect to the oracle problem, mutation analysis, and random testing.

Franz Schweiggert studied Mathematics and Computer Science at Technical University Munich (TUM) from 1970 until 1975. In 1979 he got his PhD at TUM for his thesis concerning isomorphism of graphs and discrete structures. Furthermore in 1979 he got a diploma in Business Administration. From 1980 until 1984 he worked at industry in the field of software quality assurance. Since 1984 he is Professor for Computer Science at Ulm University, since 1989 head of the Institute of Applied Information Processing.