Debugging Programs using Formal Concept Analysis

Artem Revenko[1,2]

[1] Technische Universität Dresden
Zellescher Weg 12-14, 01069 Dresden, Germany
[2] National Research University Higher School of Economics
Pokrovskiy bd. 11, 109028 Moscow, Russia
artem_viktorovich.revenko@mailbox.tu-dresden.de

Abstract. The classification of possible errors in object intents is given and some possibilities of exploring them are discussed. Two techniques for finding some types of errors in new object intents are introduced. After comparing the better technique is developed further in order to guarantee the absence of certain errors given enough information. Based on this technique an approach for debugging source code is presented and discussed. It is shown that the new approach yields bug hypothesis in a strict logical form. Using the new approach it is possible to come closer to debugging programs on a logical level not checking executions line by line. An example of applying the new approach is presented.

Keywords: formal context analysis, implication, debugging

1 Introduction

In this work we present a new approach to debug programs. This work was inspired by the Delta Debugger project [13] where authors discuss the possibilities of automatic debugging, namely isolation of failure-inducing inputs. However, when it comes to finding actual causes of the failure it is still not possible to automatically explain the failure logically. In some cases the nearest neighbour technique yields good results, but usually near-probabilistic criteria like coverage or chi-square are used [2]. In this work we use recent advance in Formal Concept Analysis in an attempt to find logical dependencies between fails and successful runs of a program. Several studies were performed to discover the possibilities of using Formal Concept Analysis in software development. For example, in [11] and [6] authors use Formal Concept Analysis for building class hierarchies. In [8] FCA is used to determine dependencies on program trace. Authors reveal causal dependencies and even are able to find "likely invariants" of program in special cases. However, they do not consider the possibility of debugging. However, to our best knowledge there are no works about applying Formal Concept Analysis to program debugging.

In this paper we first introduce a new technique for finding errors in new object intents. This technique was first introduced in our previous work; we partly

repeat the results and refer to our previous work for more details. In this paper we recall two different approaches for revealing errors in new object intents: one based on computing the implication system of the context and another one based on computing the closures of the subsets of the new object intent. Since computing closures may be performed much faster we improve and generalize this approach and finally obtain a procedure for finding all possible errors of the considered types. We also provide experimental results to compare two approaches. After that we present a new approach of debugging based on the discussed above technique of finding errors in data. An example of debugging is provided.

All sets and contexts we consider in this paper are assumed to be finite.

2 Main Definitions

Let G and M be sets. Let $I \subseteq G \times M$ be a binary relation between G and M. Triple $\mathbb{K} := (G, M, I)$ is called a *(formal) context*.
The set G is called a set of *objects*. The set M is called a set of *attributes*.

Consider mappings $\varphi\colon 2^G \to 2^M$ and $\psi\colon 2^M \to 2^G$: $\varphi(X) := \{m \in M \mid gIm \text{ for all } g \in X\}$, $\psi(A) := \{g \in G \mid gIm \text{ for all } m \in A\}$. For any $X_1, X_2 \subseteq G$, $A_1, A_2 \subseteq M$ one has

1. $X_1 \subseteq X_2 \Rightarrow \varphi(X_2) \subseteq \varphi(X_1)$
2. $A_1 \subseteq A_2 \Rightarrow \psi(A_2) \subseteq \psi(A_1)$
3. $X_1 \subseteq \psi\varphi(X_1)$ and $A_1 \subseteq \varphi\psi(A_1)$

Mappings φ and ψ define a *Galois connection* between $(2^G, \subseteq)$ and $(2^M, \subseteq)$, i.e. $\varphi(X) \subseteq A \Leftrightarrow \psi(A) \subseteq X$. Usually, instead of φ and ψ a single notation $(\cdot)'$ is used. $(\cdot)'$ is sometimes called a *derivation operator*. For $X \subseteq G$ the set X' is called the *intent* of X and is denoted $\mathrm{int}(X)$. Similarly, for $A \subseteq M$ the set A' is called the *extent* of A and is denoted $\mathrm{ext}(A)$.
Let $Z \subseteq M$ or $Z \subseteq G$. $(Z)''$ is called the *closure* of Z in $\mathbb{K}$. Applying Properties 1 and 2 consequently one gets the *monotonicity* property: for any $Z_1, Z_2 \subseteq G$ or $Z_1, Z_2 \subseteq M$ one has $Z_1 \subseteq Z_2 \Rightarrow Z_1'' \subseteq Z_2''$.
Let $m \in M, X \subseteq G$, then $\overline{m}$ is called a *negated* attribute. $\overline{m} \in X'$ whenever no $x \in X$ satisfies xIm. Let $A \subseteq M$; $\overline{A} \subseteq X'$ iff all $m \in A$ satisfy $\overline{m} \in X'$.

An *implication* of $\mathbb{K} := (G, M, I)$ is defined as a pair (A, B), written $A \to B$, where $A, B \subseteq M$. A is called the *premise*, B is called the *conclusion* of the implication $A \to B$. The implication $A \to B$ is *respected by a set of attributes* N if $A \nsubseteq N$ or $B \subseteq N$. The implication $A \to B$ holds (is valid) in $\mathbb{K}$ if it is respected by all g', $g \in G$, i.e. every object, that has all the attributes from A, also has all the attributes from B. Implications satisfy *Armstrong rules*:

$$\frac{}{A \to A} \quad , \quad \frac{A \to B}{A \cup C \to B} \quad , \quad \frac{A \to B, B \cup C \to D}{A \cup C \to D}$$

A *support* of an implication in context $\mathbb{K}$ is the set of all objects of $\mathbb{K}$, whose intents contain the premise and the conclusion of the implication. A *unit implications* is defined as an implication with only one attribute in the conclusion,

i.e. $A \to b$, where $A \subseteq M,\ b \in M$. Every implication $A \to B$ can be regarded as the set of unit implications $\{A \to b \mid b \in B\}$. One can always observe only unit implications without loss of generality.

An *implication basis* of a context $\mathbb{K}$ is defined as a set $\mathfrak{L}$ of implications of $\mathbb{K}$, from which any valid implication for $\mathbb{K}$ can be deduced by the Armstrong rules and none of the proper subsets of $\mathfrak{L}$ has this property.
A minimal implication basis is an implication basis minimal in the number of implications. A minimal implication basis was defined in [7] and is known as the *canonical implication basis*. In paper [4] the premises of implications from the canonical base were characterized in terms of pseudo-intents. A subset of attributes $P \subseteq M$ is called a *pseudo-intent*, if $P \neq P''$ and for every pseudo-intent Q such that $Q \subset P$, one has $Q'' \subset P$. The canonical implication basis looks as follows: $\{P \to (P'' \setminus P) \mid P$ - pseudo-intent$\}$.

We say that an object g is *reducible* in a context $\mathbb{K} := (G, M, I)$ iff $\exists X \subseteq G : g' = \bigcap_{j \in X} j'$.

3 Types of Errors

In this section we use the idea of *data domain dependency*. Usually objects and attributes of a context represent entities. Dependencies may hold on attributes of such entities. However, such dependencies may not be implications of a context as a result of an error in object intents. Thereby, data domain dependencies are such rules that hold on data represented by objects in a context, but may erroneously be not valid implications of a context.

In this work we consider only dependencies that do not have negations of attributes in premises. As mentioned above there is no need to specially observe non-unit implications. Consider possible types of such dependencies ($A \subseteq M,\ b, c \in M$):

1. $A \to b$
2. $A \to \overline{b}$
3. $A \to b \vee c$
4. $A \to \Phi$, where Φ is any logical formula not considered above, for example, $\Phi = a \vee (b \wedge \overline{c})$

The types 1 and 2 are most simple and common dependencies. In this work we try to find the algorithm to reveal these two types of dependencies and find corresponding errors.

4 Finding Errors

Below we assume that we are given a context (possibly empty) with correct data and a number of new object intents that may contain errors. This data is taken from some data domain and we may ask an expert whose answers are always

correct. However, we should ask as few questions as possible.

We introduce two different approaches to finding errors. The first one is based on inspecting the canonical basis of a context. When adding a new object to the context one may find all implications from the canonical basis of the context such that the implications are not respected by the intent of the new object. These implications are then output as questions to an expert in form of unit implications. If at least one of these implications is accepted, the object intent is erroneous. Since the canonical basis is the most compact (in the number of implications) representation of all valid implications of a context, it is guaranteed that the minimal number of questions is asked and no valid dependencies of Type 1 are left out.
Although this approach allows one to reveal all dependencies of Type 1, there are several issues. The problem of producing the canonical basis with known algorithms is intractable. Recent theoretical results suggest that the canonical base can hardly be computed with better worst-case complexity than that of the existing approaches ([3], [1]). One can use other bases (for example, see progress in computing proper premises [10]), but the algorithms known so far are still too costly and non-minimal bases do not guarantee that the expert is asked the minimal sufficient number of questions.

However, since we are only interested in implications corresponding to an object, it may be not necessary to compute a whole implication basis. Here is the second approach. Let $A \subseteq M$ be the intent of the new object not yet added to the context. $m \in A''$ iff $\forall g \in G : A \subseteq g' \Rightarrow m \in g'$, in other words, A'' contains the attributes common to all object intents containing A. The set of unit implications $\{A \to b \mid b \in A'' \setminus A\}$ can then be shown to the expert. If all implications are rejected, no attributes are forgotten in the new object intent. Otherwise, the object is erroneous. This approach allows one to find errors of Type 1.

5 Improvements

Obviously, applying the derivation operator two times is a much easier task than computing the canonical basis, and can be performed in polynomial time. However, the following case is possible. Let $A \subseteq M$ be the intent of the new object such that $\nexists g \in G : A \subseteq g'$. In this case $A'' = M$ and the implication $A \to A'' \setminus A$ has empty support. This may indicate an error of Type 2, because the object intent contains a combination of attributes impossible in the data domain, but the object may be correct as well. An expert could be asked if the combination of attributes in the object intent is consistent in the data domain. For such a question the information already input in the context is not used. More than that, this question is not sufficient to reveal an error of Type 1.

Proposition 1. *Let* $\mathbb{K} = (G, M, I), A \subseteq M$. *The set*

$$\mathcal{I}_A = \{B \to d \mid B \in \mathcal{MC}_A, d \in B'' \setminus A \cup \overline{A \setminus B}\},$$

where $\mathcal{MC}_A = \{B \in \mathcal{C}_A | \nexists C \in \mathcal{C}_A : B \subset C\}$ *and* $\mathcal{C}_A = \{A \cap g' \mid g \in G\}$, *is the set of all unit implications (or their non-trivial consequences with some attributes added in the premise) of Types 1 and 2 such that implications are valid in* $\mathbb{K}$, *not respected by* A, *and have not empty support.*

Proposition 1 allows one to find an algorithm for computing the set of questions to an expert revealing possible errors of Types 1 and 2. The pseudocode is pretty straightforward and is not shown here for the sake of compactness.
Since computing the closure of a subset of attributes takes $O(|G| \times |M|)$ time in the worst case, and we need to compute respective closures for every object in the context, the time complexity of the whole algorithm is $O(|G|^2 \times |M|)$.
We may now conclude that we are able to find possibly broken dependencies of two most common types in new objects. However, this does not always indicate broken real dependency, as we not always have enough information already input in our context. That is why we may only develop a hypothesis and ask an expert if it holds.
For more details, example, and proof of Proposition 1, please, refer to [9].

6 Debugging

6.1 Context Preparation

Normally debugging starts with a failure report. Such a report contains the input on which the program failed. By this we mean that our program was not able to output the expected result or did not finish at all. This implicitly defines "goal" function which is capable of determining either a program run was successful or not. We could imagine a case where we do not have any successful inputs, i.e. those inputs which were processed successfully by the program. However, it does not seem reasonable. In a such a case the best option seems to rewrite the code or look for obvious mistakes. Modern techniques of software development suggests running tests even before writing code itself; unless the tests are passed code is not considered finished. Therefore, successful inputs are at least those contained in the test suites.
As discussed in the beginning of this paper the problem of finding appropriate inputs was considered by different authors. This problem is indeed of essential importance for debugging. However, we do not aim at solving it. Instead we assume that inputs are already found (using user reports, random generator, or something else), processed (it is better if inputs are minimized, however, not necessary), and are at hands. We focus on processing the program runs on given inputs.
Our approach consists in the following. We construct two contexts: first with successful runs as objects, second with failed runs. In both cases attributes are

the lines of the code (conveniently presented via line numbers). We put a cross if during processing of the input the program has covered the corresponding line. So in both cases we record the information about covered lines during processing of the inputs. After contexts are ready we treat all the objects from the context with failed runs as new objects and try to find errors as described in the previous sections. Expected output is an implication $A \to B$. The interpretation is as follows; in successful runs whenever lines numbers A are covered, lines numbers B are covered as well. For some reason in the inspected failed run this is not the case. Debugging consists now in finding this reason. This is not absolutely automatic debugging, however, we receive some more clues and may find a bug without checking the written code line by line. More than that, this approach is strict, that is we say that it always happens, not with any probability. And it corresponds to the real situation: the bug *is* there, not with any probability.

6.2 Example

Consider the following function written in Python (example taken from [12]):

Listing 1.1: remove_html_markup [12]

```
1  def remove_html_markup(s):
2      tag   = False
3      quote = False
4      out   = ""
5      for c in s:
6          if (c == '<' and
7              not quote):
8              tag = True
9          elif (c == '>' and
10                 not quote):
11             tag = False
12         elif (c == '"' or
13                 c == "'" and
14                 tag):
15             quote = not quote
16         elif not tag:
17             out = out + c
18     return out
```

The goal of the function, as follows from its name, is to remove html markup from the input, no matter if it occurs inside or outside quotes. Therefore, we may formulate our goal as: no `<` in output. Such a formulation does not allow us to catch all the bugs (check input `"foo"` in contexts below), but it suffices for our purposes.

The function works as follows. After initialisation we have four "if" cases. The first and the second one checks if we have encountered a tag symbol outside of quotes. If so, the value of "tag" is changed. The third one checks if we have

encountered a quote symbol inside tag. This is important for not closing a tag if the closing symbol happens to be in one of the parameters (see inputs). If so, the value of "quote" is changed. The last "if" adds the current character to the output if we are outside the tag.

We consider the following set of inputs: `foo`, `<b>foo</b>`, `"<b>foo</b>"`, `"<b>a</b>"`, `"<b></b>"`, `"<>"`, `"foo"`, `'foo'`, `<em>foo</em>`, `""`, `<"">`, `<p>`, `<a href=">">foo</a>`

Using the given outputs we obtain two contexts:

Context with successful inputs	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`foo`		×	×	×	×	×			×			×	×			×	×	×
`<b>foo</b>`		×	×	×	×	×	×	×	×	×	×	×	×			×	×	×
`"foo"`		×	×	×	×	×			×			×	×		×	×	×	×
`'foo'`		×	×	×	×	×			×			×	×	×		×	×	×
`<em>foo</em>`		×	×	×	×	×	×	×	×	×	×	×	×			×	×	×
`<a href=">">foo</a>`		×	×	×	×	×	×	×	×	×	×	×	×		×	×	×	×
`""`		×	×	×	×	×			×			×			×			×
`<"">`		×	×	×	×	×	×	×	×	×	×	×			×			×
`<p>`		×	×	×	×	×	×	×	×	×	×	×	×			×		×

Context with failed inputs	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`"<b>foo</b>"`		×	×	×	×	×	×		×	×		×	×		×	×	×	×
`"<b>a</b>"`		×	×	×	×	×	×		×	×		×	×		×	×	×	×
`"<b></b>"`		×	×	×	×	×	×		×	×		×	×		×	×	×	×
`"<>"`		×	×	×	×	×	×		×	×		×	×		×	×	×	×

Fig. 1: Contexts with failed and successful runs

It is easy to notice that the only difference between failed inputs as context objects is their names.
Adding any of failed inputs to the first context yields the following implication:

$$7, 13, 15 \rightarrow 8, 11$$

What is essentially said is if we happened to cover lines 7, 13, 15, we should have also had been inside tag (lines 8 and 11). Given some thought and attention we realize that this is absolutely true, because it is not clear how we could reach line 15 without having "tag"= *true*, as this condition is checked in line 14.
In Python as well as in many other languages logical operation "and" has a higher priority as "or", so condition of the third "if" `(c == '"' or c == "'" and tag)` is implicitly transformed in `(c == '"' or (c == "'" and tag))`, that is why on lines 12 and 13 brackets are forgotten. After debugging the condition should look as follows: `((c == '"' or c == "'") and tag)` and the program runs correctly.

7 Conclusion

A technique for finding errors of two types in new object intents is presented. As opposed to finding the canonical basis of the context the proposed algorithm terminates much faster. Based on this technique an approach for debugging source code is presented. This approach is capable of finding strict dependencies between lines of source code covered in successful and failed runs. The output is a logical expression which allows to debug the source code using the logic of the program. This may get us one step closer to automated debugging.

References

1. Mikhail Babin and Sergei O. Kuznetsov. Recognizing pseudo-intent is conp-complete. *Proc. 7th International Conference on Concept Lattices and Their Applications, University of Sevilla*, pages 294–301, 2010.
2. Holger Cleve and Andreas Zeller. Locating causes of program failures. In Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors, *ICSE*, pages 342–351. ACM, 2005.
3. Felix Distel and Barış Sertkaya. On the complexity of enumerating pseudo-intents. *Discrete Applied Mathematics*, 159(6):450–466, 2011.
4. Bernhard Ganter. Two basic algorithms in concept analysis. *Preprint-Nr. 831*, 1984.
5. Bernhard Ganter, Gerd Stumme, and Rudolf Wille, editors. *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*. Springer, 2005.
6. Robert Godin and Petko Valtchev. Formal concept analysis-based class hierarchy design in object-oriented software development. In Ganter et al. [5], pages 304–323.
7. J.-L. Guigues and V. Duquenne. Familles minimales d'implications informatives résultant d'un tableau de données binaires. *Math. Sci. Hum*, 24(95):5–18, 1986.
8. John L. Pfaltz. Using concept lattices to uncover causal dependencies in software. In *Proc. Int. Conf. on Formal Concept Analysis, Springer LNAI 3874*, pages 233–247, 2006.
9. Artem Revenko and Sergei O. Kuznetsov. Finding errors in new object intents. In *CLA 2012*, pages 151–162, 2012.
10. Uwe Ryssel, Felix Distel, and Daniel Borchmann. Fast computation of proper premises. In Amedeo Napoli and Vilem Vychodil, editors, *International Conference on Concept Lattices and Their Applications*, pages 101–113. INRIA Nancy – Grand Est and LORIA, 2011.
11. Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. *SIGSOFT Softw. Eng. Notes*, 23(6):99–110, November 1998.
12. Andreas Zeller. Software debugging course. https://www.udacity.com/course/cs259.
13. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.