Debugging Program Code Using Implicative Dependencies

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. Based on the technique for finding errors in new object intents a method of debugging source code is presented. This method is capable of finding strict implicative dependencies between lines of source code covered in successful and failed runs. The output is a logical expression. Using the new method 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 method is presented. Possibilities of further development are discussed.

Keywords: formal context analysis, implication, debugging

1 Introduction

Automatic debugging is not a new topic in science and is investigated by many computer scientist. For example, in [16] authors investigate a novel method of "relative debugging" which consists in comparing particular values of data structures. In [1] and [9] authors survey different approaches to automatic debugging. In the well known work [19] the Delta Debugger tool is presented; authors introduce an approach to 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. Usually near-probabilistic criteria like chi-square are used [4]. Somehow it does not correspond to the correctness of the program as a program bug is either present or not.

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. For example, it could be that as long as a particular part of "if" statement is not covered during the program run (i.e. a particular conditional clause is not satisfied) the program runs successfully; this would mean that a bug lies probably in this particular part of the program.

Implications which can be derived from data tables (formal context) represent strong logical dependencies between attributes. We use this advantage of implications to introduce a way of debugging program code following the logic of a program.

Several studies were performed to discover the possibilities of using Formal Concept Analysis in software development. For example, in [17] and [10] authors use Formal Concept Analysis for building class hierarchies. In [13] 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. A very interesting work on fault localization is presented in [3]. However, to our best knowledge there are no works about applying Formal Concept Analysis to program debugging.

In our previous work [14] we have introduced two approaches to revealing errors in new object intents. In this paper we recall them; one is based on computing the implication system of the context and another one is 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.

After that we present a method of debugging based on the discussed above technique of finding errors in data. We provide an example and discuss the possibilities of further development.

2 Main Definitions

In what follows we keep to standard definitions of FCA [8]. Let G and M be sets and 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 $\psi\colon 2^M \to 2^G$: $\varphi(X) := \{m \in M \mid gIm \text{ for all } g \in X\}$, $\psi(A) := \{g \in G \mid gIm \text{ for all } m \in A\}$. Mappings φ and ψ define a *Galois connection* between $(2^G, \subseteq)$ and $(2^M, \subseteq)$, i.e. $\varphi(X) \subseteq A \Leftrightarrow \psi(A) \subseteq X$. Hence, 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 $A_1 \subseteq \varphi\psi(A_1)$

Usually, instead of φ and ψ a single notation $(\cdot)'$ is used. $(\cdot)'$ is usually called a *derivation operator*. For $X \subseteq G$ the set X' is called the *intent* of X. Similarly, for $A \subseteq M$ the set A' is called the *extent* of 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 [11] and is known as the *canonical implication basis*. In [6] 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 $\exists X \subseteq G : g' = \bigcap\limits_{j \in X} j'$.

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

3 Finding 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.

Every object in a context is described by its intent. In the data domain there may exist dependencies between attributes. 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. In this work we try to find the algorithm to reveal the following two most simple and common types of dependencies ($A \subseteq M,\ b, c \in M$):

1. If there is A in an object intent, there is also b, which is represented by the implication $A \to b$
2. If there is A in an object intent, there is no b, which can be symbolically represented as $A \to \overline{b}$

If we have no errors in a context, all the dependencies of Type 1 are deducible from implication basis. However, if we have not yet added enough objects in the context, we may get false consequence. Nevertheless, it is guaranteed that none of valid dependencies is lost, and, as we add objects without errors we reduce the number of false consequences from the implication basis.

The situation is different if we add an erroneous object. It may violate a dependency valid in the data domain. In this case, until we find and correct the error, we are not able to deduce all dependencies valid in the data domain from the implication basis, no matter how many correct objects we add afterwards.

We aim to restore valid dependencies and therefore correct 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 quickly recall two different approaches to finding errors introduced in our previous works. 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 even with polynomial delay ([5], [2], [12]). One can use other bases (for example, see progress in computing proper premises [15]), 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.

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 $\mathrm{O}(|G| \times |M|)$ 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 $\mathrm{O}(|G|^2 \times |M|)$.

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 the proof of Proposition 1, please, refer to [14].

4 Debugging

4.1 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 suggest 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 method 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 the processing of the input the program has covered the corresponding line. So in both cases we record the information about covered lines during the processing of the inputs. After the 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 implication of the form $A \to B$. The interpretation is as follows: in successful runs whenever lines A are covered, lines B are covered as well. However, in the inspected failed run lines A *were* covered and lines B *were not* covered. Debugging consists now in finding the reason why lines B were not covered in the processing of the failed run.

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 method is logically strict, it does not deal with any kind of probability. This corresponds to the real situation: the bug *is* or *is not* there, not with any probability.

4.2 Example

Consider the following function written in Python (example taken from [18]):

Listing 1.1: remove_html_markup [18]

```
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 (as seen from the contexts below the input `"foo"` has enabled the program to reach the line 15 which should not have happened, but it is considered as a successful run), 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>`

We run the function on every input and check if the output contains the symbol "<". If it does not, the run was successful. We also record the lines coverage during every run, gather this information, and construct two context:

Context with successful runs	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 runs	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 in the processing of every failed input the same lines are covered. Therefore, the only difference between different objects in the context with failed runs is the names of the objects.

Inspecting any of the failed inputs in the context with successful runs using the described above technique for finding errors yields the following implication:

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

What is essentially said is the following: in a *successful* run if the lines 7, 13, and 15 were covered then the lines 8 and 11 *were also* covered; in every *failed* run the lines 7, 13, and 15 were covered, but the lines 8 and 11 *were not* covered. We now expect the mentioned above lines and their impact to the program run.

If the line 7 is covered then the condition in the line 6 was met. Therefore, the line 7 is covered whenever there is the symbol “`<`” in the input.

If the line 13 is covered then the condition in the line 12 was not met and the conditions of the first two “if” clauses were not met. This means that for some symbol from the input we should not change the value of “tag” and the symbol is not “`"`”.

If the line 15 was covered then the condition of the third “if” clause was met and the conditions of the first two “if” clauses were not met. Therefore, some symbol in the input was either “`"`” or “`’`” and the “tag” was set to `True`.

If the line 8 was not covered then the value of “tag” was never set to `True` (because we do not have this variable set to `True` elsewhere in our program and originally it is initialized to `False`). There is no need to go further to the line 11 in our investigation, because we have already found a contradiction. The value of “tag” should have been set to `True` to reach the line 15, but it was never set to `True`. From this we can deduce that possibly the condition of the third “if” clause erroneously evaluates to `True` without checking that “tag” equals `True`.

The key to this puzzle is the following. In Python as well as in many other languages logical operation “and” has a higher priority than “or”, so condition of the third “if” `(c == ’"’ or c == "’" and tag)` is implicitly transformed in `(c == ’"’ or (c == "’" and tag))`. In other words 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.

4.3 Further Development

The sequence in which the lines of code were covered contains even more information about the execution of a program. This information may reveal even more dependencies in the working flow of a program. It may also happen that the only difference between successful and failed runs is in the sequence in which the lines are covered, whereas the set of covered lines remains the same. In this manner we also take into account the sequential aspect of the loop that is not taken into account in the previous example.

It is not difficult to extend the introduced method with this new feature. For this purpose we change the attributes of our context. Now an attribute contains two numbers: the first number corresponds to the preceding covered line and the second number corresponds to the succeeding covered line. The information from the original modification of the method may still be captured with attributes that have the same line number two times. However, we may be not interested in the absolute precedence relation between the lines, because this information is excessive and difficult to interpret while the size of the set of attributes increases dramatically. That is why we also introduce a new parameter containing information about the delay of interest. Below we are only interested in the precedence relation within this delay, i.e. not more than the dealy number of lines should be covered between the two specified lines in order to add them to the relation I.

The number of attributes for this case, assuming that any line may precede and succeed any other line including itself, is $|M|^d$, where d is the delay.

Unfortunately, the result obtained using this modification may be difficult to interpret even if the delay is small. It makes sense to consider this modification only if the standard modification does not yield any results.

For the `remove_html_markup` function and $d = 1$ we obtain the following results:

1. $(10,10),(12,15) \to (8,8),(11,11),(10,11),(11,5),(9,10),(7,8),$
   $\overline{(17,17)},\overline{(9,12)},\overline{(15,5)},\overline{(13,13)},\overline{(13,16)},\overline{(16,16)},\overline{(16,17)};$
2. $(17,17),(15,5) \to \overline{(10,10)},\overline{(7,7)};$
3. $(13,13),(15,15),(7,7) \to \overline{(9,12)},\overline{(16,17)},\overline{(12,15)},\overline{(15,5)},$
   $(8,8),(11,11),(10,11),(11,5),(9,10),(7,8),(12,13).$

In the result we still have the same obtained dependency as we had before, namely Implication 3. However, we have two more obtained results that reveal more structural features of the bug. An interpretation of this result is left to the reader.

For example, we consider Implication 1. Actually already in the premise of this implication we can recognize a clue to finding the bug. Indeed, having a pair 12 and 15 with delay 1 means the following: after the line 12 the execution jumped to the line 15. After checking only condition in the line 12 (which happened to be true) execution jumped to the consequence. As already described, the interpreter has understood the condition as a disjunction and has only evaluated the first expression (which would be enough in case of disjunction).

5 Conclusion

Based on the procedure for finding errors in new object intents a method of debugging source code was proposed. This method finds strict dependencies between source code coverage in successful and failed runs. The output of the debugging method is a logical expression which allows one to find bugs following the implicit logic of the program. Further modification of the method is possible (described above), however, it leads to results that are difficult to interpret.

Acknowledgements The author was supported by German Academic Exchange Service (DAAD).

Author thanks Sergei Kuznetsov and other participants of the project Mathematical Models, Algorithms, and Software Tools for Intelligent Analysis of Structural and Textual Data supported by the Basic Research Program of the National Research University Higher School of Economics for discussion and useful remarks.

References

1. Hiralal Agrawal. Towards automatic debugging of computer programs. Technical report, ph.d. thesis, Purdue University, 1991.
2. Mikhail A. Babin and Sergei O. Kuznetsov. Computing premises of a minimal cover of functional dependencies is intractable. *Discrete Applied Mathematics*, 161(6):742–749, 2013.
3. Peggy Cellier, Mireille Ducassé, Sébastien Ferré, and Olivier Ridoux. Formal concept analysis enhances fault localization in software. In Raoul Medina and Sergei A. Obiedkov, editors, *ICFCA*, volume 4933 of *Lecture Notes in Computer Science*, pages 273–288. Springer, 2008.
4. 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.
5. Felix Distel and Barış Sertkaya. On the complexity of enumerating pseudo-intents. *Discrete Applied Mathematics*, 159(6):450–466, 2011.
6. Bernhard Ganter. Two basic algorithms in concept analysis. *Preprint-Nr. 831*, 1984.
7. Bernhard Ganter, Gerd Stumme, and Rudolf Wille, editors. *Formal Concept Analysis, Foundations and Applications*, volume 3626 of *Lecture Notes in Computer Science*. Springer, 2005.
8. Bernhard Ganter and Rudolf Wille. *Formal Concept Analysis: Mathematical Foundations.* Springer, 1999.
9. Michael Gerndt. Towards automatic performance debugging tools. In *AADEBUG*, 2000.
10. Robert Godin and Petko Valtchev. Formal concept analysis-based class hierarchy design in object-oriented software development. In Ganter et al. [7], pages 304–323.
11. 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.
12. Sergei O. Kuznetsov and Sergei A. Obiedkov. Some decision and counting problems of the duquenne-guigues basis of implications. *Discrete Applied Mathematics*, 156(11):1994–2003, 2008.
13. 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.
14. Artem Revenko and Sergei O. Kuznetsov. Finding errors in new object intents. In *CLA 2012*, pages 151–162, 2012.
15. 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.
16. Aaron James Searle. *Automatic relative debugging.* 2006.

17. Gregor Snelting and Frank Tip. Reengineering class hierarchies using concept analysis. *SIGSOFT Softw. Eng. Notes*, 23(6):99–110, November 1998.
18. Andreas Zeller. Software debugging course. https://www.udacity.com/course/cs259.
19. Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Software Eng.*, 28(2):183–200, 2002.