Identifying behavior changes after PHP language migration using static source-code analysis

S. Vellinga
April 2010

Thesis Supervisor: dr. J. Vinju
Internship Supervisor: ing. J. Hofstede
Organization: The Patient Safety Company

Master's Thesis
Master Software Engineering, University of Amsterdam

Contents

Abstract

Migrating the source-code of a PHP 4 program to PHP 5 could change the source-code semantics and therefore cause unexpected behavior. The goal of this project is to identify the causes for such changed behavior and build a tool which is able to locate the source-code that is subject to these changes using static source-code analysis.

The key difference in PHP 5 compared to its predecessor is the completely rewritten OOP-model which offers better support for object oriented programming. One of the main changes in the new OOP-model is that objects are now passed and assigned by-reference instead of by-value which leads to much more aliases.

When objects are accessible through multiple paths, modifying the object at one location could influence behavior at another location if the same object is referenced. Detecting such changed behavior is difficult because PHP is a dynamic typed programming language which means type information is not available.

Because no type information is available, resolving method invocations has to be conservative. If the variable type is unknown, all classes which implement the requested method should be considered which could result in incorrect resolved method invocations.

Also PHP's include mechanism includes files at run-time and allows the use of arbitrary expressions to include files. In order to resolve function calls, static source-code analysis has to know in which files the functions can reside. Therefore the possible values of the include expressions need to be computed.

When the precise value of an include expression cannot be resolved, file resolution needs to be conservative. This could result in incorrect resolved file inclusions.

Identifying source-code which semantics are different in PHP 5 compared to PHP 4 depends on the gathered facts previously described. Incorrect facts could result in false identified source-code or in overlooked source-code which should had been identified.

This research shows it is possible to automatically identify source-code which semantics are different in PHP 5 compared to PHP 4 using static source-code analysis. We have build a tool which was used to analyze 22 unique test cases and to analyze modules from a in production source-base developed by The Patient Safety Company consisting of over 40.000 lines of source-code.

Each test case represented a unique combination of variables which could or could not lead to changed behavior. The tool did correctly identify 11 cases which would behave different in PHP 5 and correctly skipped 6 cases which would not behave different in PHP 5. However 3 cases were incorrectly skipped by the tool and two cases were incorrectly identified.

Because the tool has to make conservative assumptions, false positives could arise. However, the results on the in production source-base showed only 68 spots of source-code which could behave different in PHP 5 compared to PHP 4 which is less than 1% on a source-base of over 40.000 lines of code.

All 68 spots turned out to be false positives caused by various reasons. While false negatives have not been tested other than implementing the unique test cases the tool did correctly identify in the in production source-base, we assume they still exist since not all small test cases were identified while they should had. Because the false negatives are caused by common language structures, it is plausible they also remain in the real test case. The unique test cases were correctly identified in the in production source-base though.

Both false negatives and false positives were caused by missing facts, the tools performance could be improved by implementing methods to extract these missing facts. The author of this thesis believes it is possible to extract these facts but this is something for further work. Currently running the tool and improving the identified spots does not guarantee no source-code which semantics are different in PHP 4 compared to PHP 5 remain.

Preface

This thesis is the result of the research project I did to conclude the one year Master's education in Software Engineering at the University of Amsterdam.

I would like to thank Niels Greidanus for giving me the change to perform this research at the Patient Safety Company.

I would like to thank Jurgen Vinju for his enthusiasm and the many helpful tips he provided.

Finally I would to thank all my colleagues from the Patient Safety company for their support and help. I want to thank Marijn Koesen in particular for the debates we had on many subjects which were of great help to me.

Sander Vellinga
Heerhugowaard, 11 April 2010

Chapter 1

Introduction

1.1 Motivation

PHP, also know as PHP: Hypertext Preprocessor, is a widely used scripting language to produce dynamic web pages. In may 2000, PHP 4 was introduced. Since its introduction PHP 4 has become really popular for generating web-pages and web-applications. Popular projects such as Wordpress, Drupal, phpMyAdmin, Mantis and Elite Bulletin Board were written in PHP 4.

As of august 2008, The PHP Group has discontinued the support for PHP 4, moving their main focus to PHP 5 and the next major release of PHP, PHP 6. Developers are encouraged to upgrade their software to PHP 5 and already a lot of projects have been upgraded. However there are still large PHP 4 legacy applications left waiting to be migrated.

The Patient Safety Company develops software for health care institutions to monitor and improve their patients safety. The software is currently used by over 50 international health-care organizations and consist of over a million lines of code. The software is written in PHP 4 and since this version is no longer supported by The PHP Group, the source-base needs to be migrated to the latest stable version of PHP, which is PHP 5.

Manually performing the language migration may take a lot of effort and even introduce bugs, especially because a large source-base needs to be rewritten. To perform the migration, a programmer needs to know what has changed in PHP 5 and fully understand what these changes imply.

The changes can be divided in two categories: Backward compatible and backward incompatible changes. For this research we want to migrate the PHP 4 source-base so that it works in PHP 5. Therefore we are only interested in the backward incompatible changes, since the backward compatible changes should not effect the PHP 4 source-code.

The goal of this research is reducing the complexity of identifying source-code that needs to be migrated. So when is this complex? For this research we will consider identifying source-code that needs to be migrated complex, when its not trivial some piece of source-code should be migrated, code that is actually correct in PHP 5 but that acts different compared to PHP 4. By executing this code, unexpected behavior could occur but the program will keep running without throwing any exceptions.

To reduce the complexity of migrating a PHP 4 source-base to PHP 5, we have build a tool that tries to identify source-code that is correct in PHP 5 but behaves different compared to PHP 4. The tool uses static source-code analysis to identify possible source-code that should be migrated so no code has to be executed.

The remainder of this thesis will be used to describe and evaluate this tool and its usefulness in aiding

the programmer with a PHP 4 to PHP 5 language migration.

1.2 Scope

This research solely focuses on identifying source-code which is syntactically correct and does not produce any errors or notices but which source-code semantics are different in PHP 5 compared to PHP 4. In other words the source-code works just fine but not as intended by the original programmer. Static source-code analysis is used to identify the source-code.

In this research performance is not an issue. Although it might be important for practical usage, this research focuses on what we are able to identify, not how fast.

1.3 Research questions

The goal of this project is reducing the complexity of identifying source-code that needs to be migrated by automating this process. The main research question is:

> **What behavior changes can static source-code analysis precisely identify on a PHP 4 to PHP 5 language migration?**

Before we can answer the main research question, we need to ask two additional questions:

- What changes cause PHP 5 source-code to behave different from PHP 4?
- What facts are needed to identify source-code that behaves different in PHP 5 compared to PHP 4?

What changes are we looking for?
In order to determine what behavior changes we are able to identify with static source-code analysis, we need to find out what changes in PHP 5 cause different behavior. We also need to understand why these changes cause source-code to behave different in PHP 5. Once we have determined what changes we are looking for, we can focus on how to identify source-code that is subject to these changes.

What facts do we need?
When it is clear what changes cause different behavior in PHP 5, we need a method of identifying source-code that is influenced by these changes. This will tell us what facts we need to know to identify the source-code that needs to be migrated. It is important we can collect or generate the necessary facts, because without these facts, it should be impossible to identify source-code that needs to be migrated.

How do we identify the source-code with changed semantics?
When we've extracted the required facts, we try to use these facts to spot source-code which might behave different in PHP 4 compared to PHP 5. For each change we are looking, there should be some method of identifying source-code which is subject to the change, the source-code should satisfy certain conditions.

If we are able to extract the required facts to determine if the conditions are satisfied, we should be able to locate source-code which is subject to a certain change. The accuracy of the analysis depends on the quality of the extracted facts, the more precise the extracted facts are, the more accurate the analysis should become.

1.4 Contributions

In order to automatically identify source-code that needs to be migrated, we need to know what changed in PHP 5 compared to PHP 4. For each change that is non-trivial to identify and which causes changed behavior, we need to determine what information is needed to locate the source-code that is effected by this change. These are the 'facts' we will use to determine if some piece of source-code needs to be migrated.

When we know what facts we need, we can check whether it is possible to extract the information using static source-code analysis and if needed, generate new facts from the previous collected data. Finally a tool is build which uses the facts to identify source-code locations that need to be migrated in order to prevent changed source-code semantics.

Because PHP is a dynamic typed programming language, it is difficult to perform a static source-code analysis. Conservative assumptions have to be made which could lead to false positives or even false negatives if facts cannot be extracted from the source-code.

To evaluate the tools usefulness, different test cases are made with known locations of source-code that should be migrated. Each test case holds a different implementation of the changes we are looking for. By evaluating the tools analysis, we can determine what it is able to identify and what not.

Besides using small test cases, we also use a module from the Patient Safety Company as a real world scenario. This gives us the opportunity of evaluating the tools analysis on a realistic program. Because the module doesn't support automated tests, it would require a lot of effort to manually check if code is not broken after the migration.

The tool could reduce the complexity of migrating a large program which has no automated tests when the results are accurate. If accurate, a programmer only has to check the source-code identified by the tool instead of the whole program.

1.5 Thesis overview

The remainder of this thesis is set up as follows:

Chapter 2 describes the used approach to set up this project. Chapter 3 gives background information about used techniques while chapter 4 describes work related to this project.

Then this thesis continues with the three main chapters in thesis which answer the research questions. Chapter 5 contains the changes which will cause changed language semantics when source-code is run in PHP 4 or PHP 5. Chapter 6 then answers which information is required to identify the source-code which might behave different and how this information is obtained.

Chapter 7 explains when source-code is identified and how this process works. Also the results achieved be the tool are discussed in this chapter. Finally in chapter 8 a summary of the project and the final conclusion is given which answers the main research question.

The appendices contain the following information:
Appendix A contains the changes between PHP 4 and PHP 5.
Appendix B describes the algorithm used by the analysis to calculate transitive closure.
Appendix C contains the database model used by the tool.
Appendix D gives an overview of the test cases and achieves results.
Appendix E contains the source-code used by the small test cases.

Chapter 2

Research method

This chapter describes the main problem and the goal of this research. We will describe what we are going to research and how we are going to research this. We will also describe how we want to validate our results.

2.1 Goal

The goal of this project is reducing the complexity of identifying source-code that needs to be migrated by automating this process. This research solely focuses on identifying source-code which is syntactically correct and does not produce any errors or notices but which source-code semantics are different in PHP 5 compared to PHP 4.

2.2 Method

To answer the main research question *"What behavior changes can static source-code analysis precisely identify on a PHP 4 to PHP 5 language migration?"* Several steps will be followed:

- Changes that cause different behavior in PHP 5
- Required facts from the source-code
- Identifying source-code that behaves different in PHP 5
- Validation

Next, each step will be described in more detail. We explain what work needs to be done in a step and what the expected result will be for that step.

2.2.1 Changes that cause different behavior in PHP 5

Before we are able to identify source-code that needs to be migrated, we need to know what we are looking for. The first step is going through the differences between PHP 4 and PHP 5 to determine what changes cause source-code semantics to be different. A list of differences between PHP 4 and PHP 5 has been provided by The PHP Group and can be retrieved from their website. These changes are described in appendix A.

2.2.2 Required facts about the source-code

For changes which cause source-code semantics to be different in PHP 5 compared to PHP 4, we have to know under what circumstances a change actually causes changed behavior and how to recognize this in the source-code. We try to express these circumstances as conditions which need to be satisfied so we can use these conditions to determine if source-code might behave different in PHP 4 compared to PHP 5.

The information required to determine if the conditions are satisfied, are the facts we need to extract from the source-code. These facts are extracted from the source-code using static source-code analysis, for this we have written a plugin for a tool which can be used to analyze PHP programs, also see section 3.2 for a description of this tool.

Not all facts can be extracted from the source-code directly, some need to be calculated from the previous collected facts. For example alias information or a function call graph cannot be retrieved directly from the source-code instead this information is calculated by our own tool based on the previous extracted facts.

2.2.3 Identifying source-code that behaves different in PHP 5

Finally the source-code is identified using the facts previously described. Because the analysis uses the facts to determine if certain conditions are satisfied in order identify source-code, the analysis result depends on the quality of the extracted and calculated facts.

Because PHP is a dynamic programming language, the analysis has to make conservative assumptions which makes the facts less precise and therefore could have a negative influence on the analysis final result.

2.2.4 Validation

To evaluate the tool, two different sorts of test cases are used. First small test cases will be set up. The test cases consist of PHP 4 source-code which either has the same behavior in PHP 5 or which behaves different in PHP 5. The test cases are unique implementations of source-code that behave different and should provide insight in when the tool is able to identify the source-code and when not.

Besides small test cases, a real program is used to evaluate the tool. The requirement for the module is that it is large, but not to large, around 40.000 lines of code.

2.3 Measurements

The perfect tool would find all and only source-code which behaves different in PHP 5 compared to PHP 4. In other words: there are no false positives and no false negatives. This can be measured with precision and recall. Precision is a measure of exactness and recall is a measure of completeness. This section describes how we will measure and evaluate the tool.

2.3.1 Evaluation

To evaluate the tools analysis, we measure the precision and recall based upon what source-code the tool is able to identify and what source-code the tool fails to identify.

Precision
To measure precision, we use the results from the test cases to see whether the source-code which the tool identifies as code that behaves different, truly behaves different in PHP 5. Below the formula for calculating precision is given where 'tp' stands for true positives and 'fp' stands for false positives.

$$\text{Precision} = \frac{tp}{tp + fp}$$

Recall
To measure recall, we use the results from the test cases and check if the tool correctly identifies all source-code which we know will behave different. Below the formula for calculating recall is given where 'tp' stands for true positives and 'fn' stands for false negatives.

$$\text{Recall} = \frac{tp}{tp + fn}$$

2.3.2 Threats to validity

Our method of evaluating the tools results might introduce threats to validity. Possible threats to validity are described below:

- PHP migration guide not complete
- Incomplete test cases
- Checking results on real test case is human work

PHP migration guide not complete

The PHP Group has released a PHP 4 to PHP 5 migration guide on their website. This guide describes all the changes between PHP 5 and PHP 4 and how to deal with them. We use this guide for the changes we will investigate to check if they will cause changed behavior. When the guide is incomplete, we could miss changes which cause changed behavior.

Incomplete test cases

Because we designed the small test cases ourself, it could be possible the cases are made to be found by the tool. While this could be true, the small test cases still tell us what source-code can be found and what source-code will not be found.

Knowing what source-code the tool is able to identify and what source-code the tool will not identify, enables us to predict what source-code we can identify. Rather than guessing if the tool will identify the source-code, we can base on facts the tool will or will not identify the source-code. Note this only accounts to source-code that has a test case representation.

Also, knowing what source-code the tool fails to recognize, gives us the opportunity of studying the problem. Why doesn't the tool identify a particular piece of source-code? Perhaps we can come up with a solution and improve the tool.

To give the reader more insight in the quality of the small test cases, the cases are included in appendix E. This allows the reader to decide for them self if the cases represent common usage of PHP source-code and maybe even to come up with additional test cases.

Now there is still the possibility our small test cases do not cover all possible occurrences of changed behavior introduced by the changes. If the test cases are not complete, two things can happen. We either miss source-code the tool is able to identify or we miss source-code the tool is not able to identify. While it is interesting to know if the tool is able to identify other source-code, the latter is of more importance.

To minimize the possibility of overlooking source-code, a real case is used. The Patient Safety Company develops software for health care institutions to monitor and improve their patients safety. The main product is a Clinical Risk Management Suite, which consist of several components called modules. One of these modules will be used as a realistic test case for the tool.

By checking the results on the real test case, we might find new source-code the tool can also correctly identify. To test if there is other source-code the tool fails to identify, we will implant variations of the small test cases in the real case and see if the tool will identify these implementations.

Checking results on real test case is human work

The identified spots in the real test case are checked by a programmer which in this case is the author of this thesis. It is possible mistakes are made and the author might be biased. For both reasons it would be better if the results were checked by other neutral programmers who had no relation to the project.

Because of limited resources this is not possible and therefore should be considered as a threat to validity.

Chapter 3

Background and context

In this chapter we describe the techniques used in this thesis and provide background information. The main topics which will be described are:

- PHP Characteristics
- phc - the open source php compiler
- Program analysis

3.1 PHP Characteristics

PHP is a dynamic typed programming language, this means PHP does not require the explicit declaration of the variables before they're used. Because there is no explicit variable declaration, variable types are determined at run-time. Since we will use static source-code analysis, this affects our analysis. The analysis becomes less accurate because one has to conservatively approximate a variable's type.

Consider the following example:

```
function setGrade($student, $grade) {
    [...]
}
```

The function 'setGrade' has two parameters, but what are their types? By using static-source code analysis, we cannot tell what types the parameters are. Why is this important? Because if a method invocation happens, we don't know what method is called. Look at the following example:

```
$student->setName($name);
```

Apparently 'student' is an object, since some method is being called on the variable. But what is the objects type? We need to know its type in order to determine what function gets called because 'getName' could exist in any class.

Also PHP's include mechanism includes files at run-time and allows the use of arbitrary expressions to include files. In order to resolve function calls, static source-code analysis has to know in which files the functions can reside. Therefore the possible values of the include expressions need to be computed.

When the precise value of an include expression cannot be resolved, file resolution needs to be conservative. This could result in incorrect resolved file inclusions which makes the analysis less accurate.

3.2 PHP Compiler

In order to collect facts from PHP source-files, a tool which is able to analyze the source-files is needed. To read the PHP source-files, an open-source PHP compiler written in C++ called phc[1] will be used. The reason phc is chosen, is that it is able to extract all required facts and that it is well documented.

When phc reads a PHP script, it builds up an internal representation of the script. The internal representation is called an Abstract Syntax Tree (or AST for short). By writing a plugin for phc, it is possible to extend the tool and use its functionality to easily walk through the AST and retrieve information about the source-code. Take a look at the following PHP program:

```
<?php
   echo 'Hello world';
?>
```

When phc reads this script, it will build up the AST as illustrated in figure 3.1.

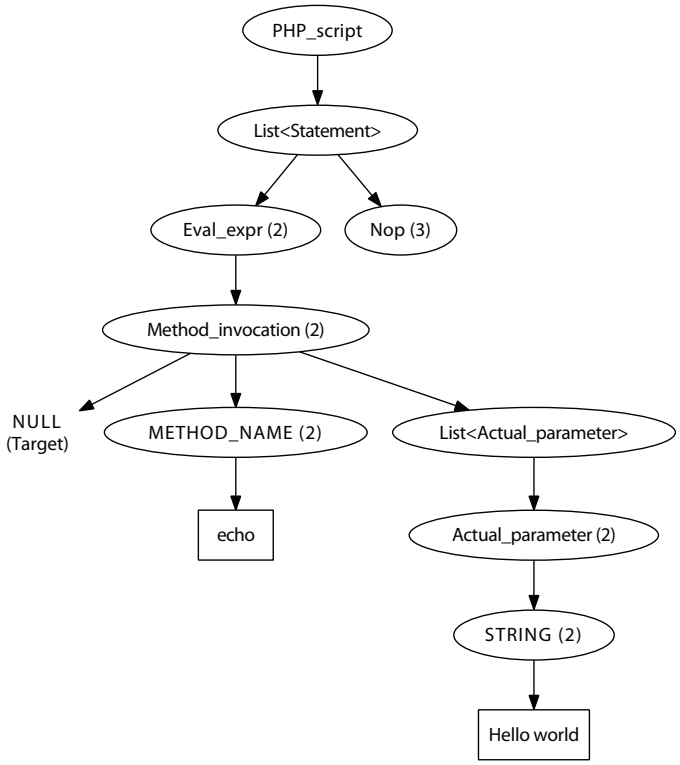


Figure 3.1: Internal phc representation of PHP program (AST)

There is a standard tree traversal class defined in phc called AST::Visitor which contains methods for each type of node in the tree. The Visitor class will automatically traverse the tree and call the associated methods for each node.

[1] http://phpcompiler.org

Actually there exists two methods for each type of node. A method which gets called before the children of the current node are visited and a method which gets called after the children of the current node are visited. By writing a plugin for phc, it is possible to extend from the Visitor class, and override the pre and post methods for each type of tree node.

3.3 Static program analysis

In this section program analysis is described which is a technique to determine the properties of a program. Program analysis can be static or dynamic but only static program analysis is described here since this technique is used by the tool developed for this project.

In contrast to dynamic program analysis, static program analysis does not require the source-code to be executed. Instead the programs properties are determined by analyzing source-code only.

3.3.1 Alias analysis

This section describes all necessary preliminary knowledge that allows us to construct a precise analysis later in chapter 6. Alias analysis is used to determine if a memory location can be referenced to by different variables called aliases. Landi and Ryder [LR92] define an alias as follows:

> "An alias occurs at some program point during program execution when two or more names exist for the same location."

Alias information can be used for compiler optimizations and program understanding for which in this thesis the latter is the goal. Different alias analyses use different properties which influence the efficiency and precision of the calculated result.

Usually there is a trade-off between the precision and efficiency [CBC93] of an algorithm in that increasing the precision of an algorithm comes at the cost of a less efficient algorithm and vice-versa.

Different analyses exist to calculate aliases and many implement a different set of properties of alias analysis. The worst-case time complexity of these analyses differs from almost linear time [Ste96] to doubly exponential time complexity in [SRW98].

Alias analysis is related to points-to analysis [EGH94] and pointer-analysis [HP00] which from now on will all be addressed as alias analysis. All these analyses are concerned with analyzing if memory locations can be accessed through multiple access paths.

Next the main properties of alias analysis are described.

Intra-procedural versus inter-procedural alias analysis

Alias analysis can either be intra-procedural or intra-procedural. Intra-procedural alias analysis does not consider aliases created at function calls and method invocations while inter-procedural alias analysis does. Intra-procedural alias analysis only considers aliases created inside a procedure which is called a function in PHP.

PHP is a dynamic typed programming language which means variables do not require an explicit declaration of the variable and its type. This also means that except at the function where a variable is instantiated, its type is unknown. Because a variable's type is only known at the location it was instantiated, inter-procedural aliases must be calculated in order to determine variable types.

Flow sensitivity

An alias analysis algorithm is flow sensitive if it considers control-flow information and the statement order of variable assignments. When a variable is reassigned all alias relations to that variable could be removed, the alias relations are "killed", this is called kill information.

A flow sensitive alias analysis would determine the alias relation only holds until it gets killed while a flow insensitive alias analysis would not. Because the flow insensitive analysis does not use control flow information it is more efficient but can be less precise.

Two variables which are aliases can be classified as must-aliases at statement S if with all execution instances of S, the same storage location is referenced. The variables which are aliases are may-aliases at S if at some execution instances of S the same storage location is referenced [CG93].

Conditional structures could cause may-aliases because at some execution of a conditional structure a variable might refer to the same storage location as another variable while at different execution of the conditional structure the alias relation might not occur.

The tool developed during this project uses a flow insensitive alias analysis and therefore calculates only may-aliases. The analysis does not distinguish executions in which variables are not aliases, if an alias relation exists at some possible execution of the program it will assume the alias relation always exists. The alias analysis used by the tool is described in section 6.4.5.

Context sensitivity

A function F can be called from different locations, when there is some call site C1 and another call site C2 which both refer to F, alias information from one caller could be leaked into the other.

Context sensitive inter-procedural alias analysis distinguishes multiple calls to the same function while context insensitive analysis treats multiple calls to the same function as a single call. Although context sensitive analysis could result in greater precision, for some benchmarks the use of context sensitive analysis show little or no improvement of precision [Ruf95].

The tool developed during this project uses a context insensitive inter-procedural alias analysis and therefore does not make a distinction between different caller sites.

Representation

Alias relations can be represented in different ways. Alias pairs store alias relations as tuples of variables, if for example variable b is an alias for a, the alias relation could be stored as <b, a>. Now these alias pairs have different representations in alias analysis algorithms.

Complete alias pairs

When complete alias pairs are used, all aliases are stored explicitly such as in [LR92]. An alias relation is symmetric, this means when variable 'b' is an alias for 'a', variable 'a' is also an alias for 'b'. This results in the alias pairs <b,a> and <a,b> which both are stored when using complete alias pairs as would the transitive alias pairs if there were any.

Compact alias pairs

When compact alias pairs are used, only basic pairs are stored such as in [CBC93]. This representation is similar to points-to representation [EGH94]. Thus the symmetric relations are not stored nor are the transitive pairs stored.

The tool developed during this project stores both complete pairs and compact pairs in separate tables because both are required for different analyses, this is explained in section 6.4.6.

3.3.2 Type analysis

Because PHP 4 is a dynamic programming language and does not use type annotations, type-inference algorithms are required to calculate type information. The common type inference algorithm is now commonly referred to as the HindleyMilner algorithm which was one of the first algorithms to calculate type-inference [Mil78].

Another example of a type-inference algorithm is the Cartesian Product Algorithm (CPA) which was originally developed for SELF [Age95]. But the Cartesian Product Algorithm can also be used to analyze different programming languages, Hanov implemented the algorithm with a few additions to analyze zscript, a dynamic typed object-oriented programming language [Han].

The Cartesian Product Algorithm is based on an algorithm by Palsberg and Schwartzbach [PS91]. Their algorithm is a constrained-based algorithm which exists of the following three basic steps:

Initialize type variables

In the first step each variable and expression in the target program get an associated type variable which represents the types an expression can hold. In the first step the type variables are only associated, in the next two steps they are filled with possible types.

Seed type variables

In the second step the analysis tries to capture the initial state of the program by adding the original types of variables and expressions, the type variables are seeded. As example, for the assignment $=10, the type variable for $x is integer.

Determine possible types

The final step builds a graph where type variables are represented with nodes. When the program executes the assignment $x = expression, the possible return types of expression must be added to the type variable of $x. Because each variable and expression is represented with a type variable, the algorithm can add an edge between the nodes in question and propagate the types of expression to the type variable of $x.

Adding edges to the graph can influence other established edges, therefore this step has to be repeated until no further propagation of types occurs.

Analysis used in this project

The tool developed during this project isn't as precise as the previous described algorithm. In this project only information about object types is needed thus default types such as strings and integers are not stored. Besides only the variable assignments which involve direct object instantiation are stored.

If the type of some variable could be the result of an expression, the analysis records a possible unknown type for the variable. Type information is used for call graph extraction. When the call graph analysis encounters a variable with an unknown type, it will ignore type information which makes the call graph extraction less precise.

3.3.3 Call graph construction

Call graphs represent relations between entities in a program. These entities can be files, modules, functions etc. In this project call graph construction is used to represent function call in a PHP program and file includes made by some file.

A large number of call graph construction algorithms exists, each with different trade-offs between analysis efficiency and call graph precision [GC01]. Below some basic approaches to calculate function call graphs are described.

Name Based Resolution (RA)

Name Based Resolution is a Reachability Analysis (RA) which only takes method names into account in order to calculate the the call graph. A more advanced version of this algorithm could also use the method signature instead of the method name.

A variation of this approach is used in [Sri92] to detect unreachable procedures in source-code either because they are never called or because their predecessors are never reached by the program.

Because only method names are used calculate the call graph, the analysis cannot distinguish methods with the same name while it is possible multiple classes use the same common names. Therefore RA is not accurate enough for our analysis.

Class Hierarchy Analysis (CHA)

The previous approach can be extended by considering subclasses which inherit methods from their parent class. This is called Class Hierarchy Analysis [DGC95]. When some class method M is reachable from function F, this also means the subclasses which inherit M are reachable.

Rapid Type Analysis (RTA)

CHA can further be extended by considering instantiated objects during the run of a program, this is know as Rapid Type Analysis [BS96]. By keeping track of variable types, only methods which exist in previously instantiated classes have to be considered.

An example project which uses this type of call graph construction is JAX [TLSS99] which is used to reduce the size of Java programs.

Analysis used in this project

The tool developed during this project uses a RTA like approach although Class Hierarchy Analysis is not performed which could produce false positives.

A RTA like approach uses the instantiated class information of variables but this information might not yet be complete because a variable could be passed to another function where its type would be unknown. Then when a method is invocated on that object, the analysis cannot use the class information.

So there is a cycle between the inter-procedural alias analysis and the call graph construction which can only be broken by making some initial assumption about the source-code. The analysis used in this project does not use the type information in its first run, and assumes a method invocation can refer to any type which contains the method.

When the call graph has been constructed inter-procedural alias analysis is performed which enables the analysis to calculate more precise method calls which in return enables the analysis to calculate more precise aliases etc. Therefore both analyses have to keep running until a fixed-point is reached.

Chapter 4

Related Work

In this chapter work related to our research is described. While source-code analysis serves many purposes [Bin07], we describe three areas which are most relevant to our research.

The main reason for changed program semantics is that in PHP 5 function arguments are passed by-reference while in PHP 4 arguments are passed by-value. Therefore one of the areas we focus on in this chapter is the detecting of references and possible conflicts.

Because PHP is a scripting language, we also focus on the analyses of other scripting languages. Web applications are often targets for attacks which makes analyses to automatically detect vulnerabilities in web applications desirable.

Finally source-code analysis is often used for program understanding. Because we use tool for program understanding such as phc to extract facts about the source-code, the last section focuses on this area.

4.1 Detecting of references and possible conflicts

In PHP 5 function arguments are passed by-reference while in PHP 4 arguments are passed by-value. This means that if an object is passed from function a to function b, both functions access the same object. Because in PHP 4 objects are passed by value, both function would not be able to access the same object since function b uses a copy of the original object.

Now in PHP 5 when one function changes the object, the other function could be affected by this change if the behavior differs when the object is changed, we call this a side effect. Different research to side effects caused by aliases has been presented in the past.

Banning tries to identify side effects which are caused by procedure calls [Ban79]. He calculates how the execution of a procedure call might effect the calling procedure. Or in other words what variables might be influenced by the execution of some procedure call.

These side-effects exists because the parameter passing mechanism can cause two variables to point to the same memory location which means they are aliases. The main difference with our research is that the side-effects we are researching are the result of language evaluation and not inherent to the original programming language.

Other research to side effects has been done by Ryder et al. They present an inter-procedural modification side-effects analysis for C that obtains better than worst-case precision on programs with general-purpose pointer usage [RLS+01].

And Larus and Hilfinger try to identify conflicts between references which occur when both references access the same field and at least one modifies the location [LH88]. This is similar to our research but instead of locating if a field is accessed twice, we check if the whole object is accessed twice and thus is less precise.

Their research focuses on Lisp-like languages where arguments are passed by value while we also have to focus on conflicts which could occur because of argument passing by-reference.

After calculating the aliases which are represented in what they call an alias graph, they try to identify conflicts between statements. In contrast to our research, they make a distinction between a statement which reads a location and a statement which writes the location.

Then with information from the alias graph they check if there are some read statements which access locations which have been accessed by write statements.

Much more research has been done on pointer analysis but this is beyond the scope of this thesis. Hind gives an extended overview of the performed research and provides many useful reference for the interested reader [Hin01].

4.2 Static detection of web application vulnerabilities

Different static source-code analyses for web applications exists. Many are used to automatically detect vulnerabilities in the source-code.

Web applications and websites are popular targets for attacks. A common type of such an attack is SQL injection where the attacker inserts SQL code as user input to be executed on the database if not probably checked. Wassermann and Su provide an analysis to detect such vulnerabilities [WS07].

While slightly similar, their analysis only considers source-code which can be influenced by an user, this is called tainted data. Many of these analysis for PHP search for such vulnerabilities while our research focuses on problems caused by changed source-code semantics instead of malicious users.

Another program which tries to locate tainted data is Pixy [JKK06a], an analysis program used to detect cross-site scripting and other vulnerabilities. Because aliases can also hold tainted values, Pixy performs an advanced alias analysis designed for PHP 4 [JKK06b].

The authors of Pixy state that a straightforward adoption of known alias analysis solutions is at least questionable because these are targeted at non-scripting languages such as C and Java. The calculate which variables are may or must aliases using alias pairs while our research does not distinguish between must and may aliases.

However, their analysis is bases on default types such as integers and string and do not support object types while our analysis is soly focused on object types. A similarity is that both analysis do not support arrays.

4.3 Program optimization and understanding

Besides analyses used for the detecting of vulnerabilities due to unexpected side-effects or tainted data, the analyses can also be used for program optimization or program understanding. Program understanding tools try to provide insight in the internal structure and behavior of some program and program optimization tries to increase the performance of the source-code.

Analysis such as pointer analysis, aliasing and data-flow analysis are often applied in compilers for static typed programming languages such as C [ASU06]. However, these analysis are more difficult to

perform on dynamically typed languages such as PHP.

Because PHP lacks type information, type errors cannot be automatically detected before the code is executed since the type is determined at run-time. It is also more difficult to perform code optimizations because the required analyses are less accurate. Besides PHP is weakly typed which means a variable could be a different type in different program execution paths.

A tool which actually can compile PHP source-code is phc [dVG07] [Big09]. This tool compiles PHP source-code into an optimized executable and combines alias analysis, type-inference and constant-propagation for PHP, to calculate the results required for its optimizations.

Besides optimizing PHP source-code, phc can also be used to pretty-print PHP code and even allows you to build tools which use phc to analyze, modify or refactor PHP scripts using C++ plugins.

The phc tool build up an internal representation of some program called an Abstract Syntax Tree (AST) (see chapter: 3.2) and then allows you to separately visit each node in the tree to either analyze the properties of your source-code or change properties of the tree which allows you to perform source-code transformations.

As described in chapter 3.2, phc is used by our research to extract the facts required for our analysis.

Chapter 5

Change in behavior

This chapter describes what changes in PHP 5 cause source-code to behave different from PHP 4 and what changed we will try to identify with a tool. The following research question will be answered:

What changes cause PHP 5 source-code to behave different from PHP 4?

5.1 The changes

Before we are able to identify source-code that needs to be migrated, we need to know what we are looking for. The PHP Group has released a PHP 4 to PHP 5 migration guide on their website. The guide describes all the changes between PHP 5 and PHP 4 and how to deal with them.

Appendix A lists all the changed mentioned in the guide in which we are looking for non fatal changes which are incompatible with PHP 4 and are not trivial to identify.

This means we are looking for changes that won't necessarily cause the program to stop running, but might cause changed behavior. Besides the source-code that is subject to this change should not be trivial to identify.

So when is source-code that needs to be migrated trivial to identify? In this research we consider behavior changes of a standard PHP function or keyword trivial to identify because we can simple search the source-base for the changed function or keyword.

5.1.1 Possible changes which could cause changed behavior

Next all language changes which qualify as possible cause for changed behavior will be named. For each change we will determine if the tool needs to identify source-code which is effected by the change.

Because we are only interested in non fatal and incompatible changes, only changes 4 till 14 are considered.

Changes

Change: objects are passed and assigned by-reference instead of by-value	
ID:	4
Identify:	Yes
Rationale	Probably the change which has most effect on PHP 4 source-code which is run in PHP 5. Because objects are assigned and passed by reference instead of by value, many more aliases will be generated. When an alias gets modified somewhere, it could effect other aliases throughout the program. If a modified alias effects other aliases depends on many things, which makes identifying source-code which will be affected by this change difficult.

Change: an object with no properties is no longer considered "empty"	
ID:	5
Identify:	No
Rationale	If the function empty() has an object with no properties as argument, this object wont be considered empty in PHP 5 while in PHP 4 it would. For this research this change wont be considered because source-code which used the empty() function can be easily identified, with linux grep for example.

Change: get_class(), get_parent_class() and get_class_methods() return names case sensitive	
ID:	6
Identify:	No
Rationale	For this research this change wont be considered because source-code which used these functions can be easily identified, with linux grep for example.

Change: in some cases classes must be declared before use	
ID:	7
Identify:	No
Rationale	This only happens if some of the new features of PHP 5 (such as interfaces) are used. Otherwise the behaviour is the old. We will assume our old PHP 4 code doesn't use PHP 5 features.

Change: include_once() and require_once() first normalize the path	
ID:	8
Identify:	No
Rationale	For this research this change wont be considered because source-code which used these functions can be easily identified, with linux grep for example.

Change: strrpos() and strripos() now use the entire string as a needle	
ID:	9
Identify:	No
Rationale	For this research this change wont be considered because source-code which used these functions can be easily identified, with linux grep for example.

Change: ip2long() now returns FALSE	
ID:	10
Identify:	No
Rationale	For this research this change wont be considered because source-code which used the ip2long() function can be easily identified, with linux grep for example.

Change: array_merge() was changed to accept only arrays	
ID:	11
Identify:	No
Rationale	For this research this change wont be considered because source-code which uses array_merge() can be easily identified, also an E_WARNING will be thrown.

Change: PATH_TRANSLATED server variable no longer set implicitly under Apache2 SAPI	
ID:	12
Identify:	No
Rationale	For this research this change wont be considered because source-code which uses the PATH_TRANSLATED server variable can be easily identified.

Change: T_ML_COMMENT constant is no longer defined by the Tokenizer extension	
ID:	13
Identify:	No
Rationale	For this research this change wont be considered because source-code which uses the T_ML_COMMENT constant can be easily identified, also when error_reporting is set to E_ALL, PHP will generate a notice.

Change: CLI version will now always populate the global $argc and $argv variables	
ID:	14
Identify:	No
Rationale	For this research this change wont be considered because source-code which uses these variables can easily be identified, with linux grep for example.

5.2 Changes to identify

From the previous section we can conclude the focus will be on identifying source-code which is affected by the change that objects are now assigned by-reference instead of assigned by-value and that objects are now passed by-reference instead of passed by-value. These are basically instances of the same problem and this section will provide more details about the problem.

5.2.1 Changed language semantics

In PHP 4 objects are assigned by-value and passed by-value. This means that whenever an object is assigned to a new variable or passed as argument to an external function, at least "logically" a copy is made, it is not necessarily done, but the observable behavior is like this. Changes made in the external function or to the new variable will only affect the copy and not the original object.

In PHP 5 this changed. Objects are now assigned and passed by-reference. It is possible to run PHP

5 in a backwards compatible mode by setting the boolean 'zend.ze1_compatibility_mode' in the PHP configuration file to true. However the main change in PHP 5 is the new OOP model, so what would be the point of upgrading to PHP 5 when not using its main new feature? Also this will have severe repercussions on run-time efficiency.

When an object is assigned to a new variable or passed to an external function, a reference to the original object is given, making the new variable an alias. Any change made to the alias is actually performed on the original object. This could cause problems when the original object (or another alias) is used after the alias is modified.

Lets see how this works, first we assign some object 'variable1' to another variable 'variable2'.

```
$variable2 = $variable1;
```

Or we pass 'variable1' as an argument to function 'f()'.

```
f($variable1);

function f($variable2) {
    [...]
}
```

In PHP 4 'variable2' would be a copy while in PHP 5 'variable2' would be an alias. This is shown in figure 5.1.

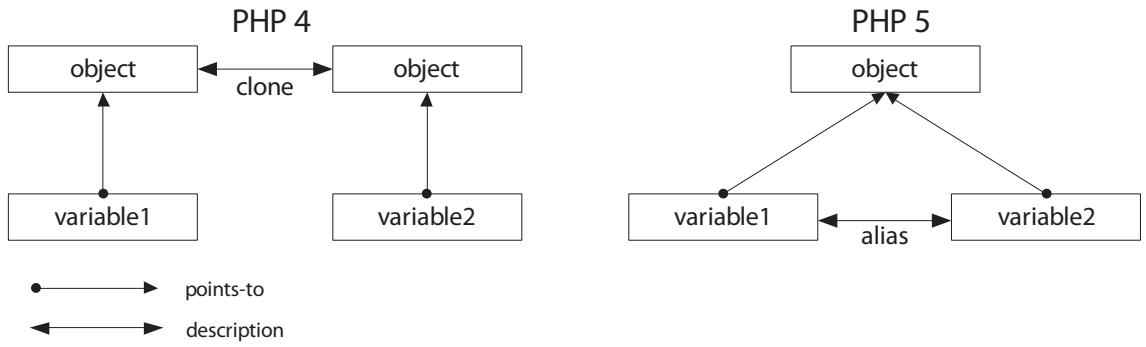


Figure 5.1: Object assigning/ passing in PHP 4 and PHP 5

5.2.2 Examples

Below are two examples of changed behavior caused by the change. First an example of assigning by reference is given, second an example of passing an argument by reference is given.

```
$personA = new Person('first');
$personB = $personA; // $personB: alias or copy?

$personB->setName('second');

echo $personA->getName();
```

The above example will print 'first' when run in PHP 4 but when run in PHP 5 it will output 'second'. This is because in PHP 4 personB is a copy while in PHP 5, personB is an alias. In PHP 5, executing

the method setName() on personB affects personA because both personA and personB refer to the same object.

```
$person = new Person('Ian');

changePersonName($person); // $person: alias or copy?

echo $person->getName();

function changePersonName($person) {
        $person->setName('James');
}
```

This example will print 'Ian' when run in PHP 4 but when this code is run in PHP 5, it will output 'James'. This is because in PHP 4 passing the variable 'person' to the function 'changePersonName' creates a copy but in PHP 5 an alias is created.

5.2.3 Changed program semantics

The behavior of the program changes when an object that has been modified has an alias which is used after the modification. Also the alias relation has to exist at the moment of modification. This is what our tool will be looking for, multiple used variables which get modified and reused after.

This means the following conjunction of conditions has to be satisfied:

- Aliases of objects which are used more than once in a function, and
- An earlier usage of one of the aliases modifies the object, and
- The alias relation exists at the moment of modification

Now lets take a look at the following source-code in figure 5.2 which shows when source-code should be identified.

```
function f($x) {
        $x2 = $x;
        $x->setProperty('changed');

        echo $x2->getProperty();
}
```

```
function f($x) {
        $x2 = $x;
        $x->setProperty('changed');

        echo $x2->getProperty();
}
```

```
function f($x) {
        $x->setProperty('changed');
        $x2 = $x;

        echo $x2->getProperty();
}
```

Figure 5.2: When behavior changes or not

In the left example of figure 5.2 the variable 'x' is used multiple times, first the method 'setProperty' is invocated on 'x' and later the method 'getProperty' is invocated on an alias of 'x'.

Lets assume 'setProperty' does what its name implies, setting some property and therefore modifies the object. This satisfies our second rule so now we only need to check if the alias relation exists at the moment of modification.

In the left example 'x2' is assigned to 'x' before 'x' gets modified while in the right example 'x2' gets assigned to 'x' after 'x' is modified. In the left example the behavior would be different in PHP 5 compared to PHP 4 but in the right example, the behavior would be the same.

5.2.4 Preventing changed program behavior

By cloning objects which are used multiple times, changed behavior can be prevented. In PHP 5, the clone() function simulates the old PHP 4 behavior by creating a copy of an object instead of an alias. However while PHP 4 makes a deep copy of an object when passed to another function or when assigned to another variable, PHP 5 only makes a shallow copy.

The difference between a shallow copy and a deep copy is that if an object has any properties which can also be objects, the properties will also be cloned with a deep copy but with a shallow copy any properties that are references to other variables, will remain references.

To really simulate the cloning behavior in PHP 5 as it would be in PHP 4, one would have to implement the method "__clone()" which is invoked once a clone is made of the object in question. From this method one could also clone the properties and if these properties contains any references to other object, these steps need to be repeated.

In the example given earlier, the PHP 4 behavior can be achieved by making the following adjustment:

```
$personB = clone($personA);
```

By cloning personA, personB is a whole new object with the same properties of personA. Now modifying personB won't affect personA.

Chapter 6

Static source-code analysis on PHP

In order to identify source-code which needs to be migrated, information about the source-code is needed. Static source-code analysis will be used to obtain this information from the source-code.

This chapter describes the various analyses required in order to collect the necessary facts and answers the following research question:

> **What facts are needed to identify source-code that behaves different in PHP 5 compared to PHP 4?**

6.1 Analysis phases

The analysis exists of three phases, in the first phase facts are extracted from PHP-source files. In the second phase the collected facts are used to generate facts which cannot be retrieved from the source-code directly. Finally in the third phase the actual analysis to identify reference problems in PHP-source files is performed.

Figure 6.1 illustrates the distinct phases together with the input and output of each phase.

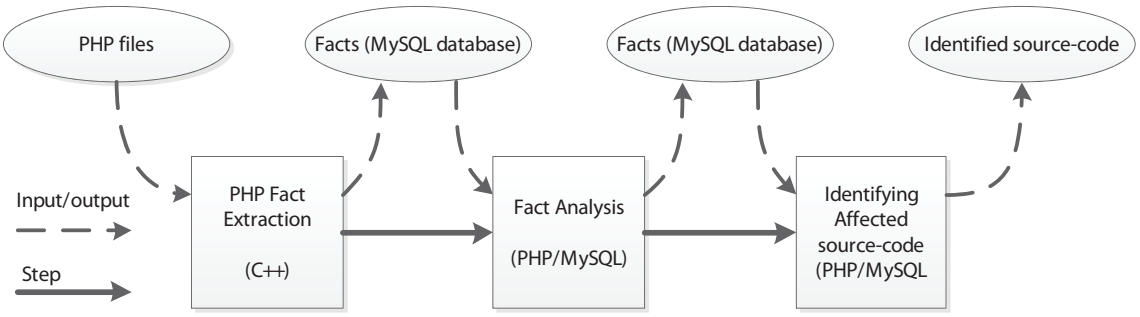


Figure 6.1: The analysis phases

In order to analyze PHP files a plugin for phc, an open-source compiler for PHP, is build. The tool extracts all required facts from the source-code and stores the data in a database. The database model can be viewed in appendix C.

In the second phase a tool written in PHP will analyze the collected facts in order to generate facts we

cannot retrieve from the source-code directly such as a call-graph extraction and alias naming. The newly generated facts will be stored back to the database.

Finally in the third phase a tool written in PHP uses the previous collected and generated facts to identify source-code which might behave different in PHP 5.

The phase 'PHP Fact Extraction' is described in section 6.3 and the phase 'Fact Analysis' is described in section 6.4. The final phase 'Identifying Affected source-code' is described in chapter 7.

6.2 Required facts

Before the PHP source-files will analyzed, the required information from the source-code has to be determined. The source-code to be identified has to satisfy the conditions described in chapter 5. This means the information needed to determine if the conditions are satisfied are the facts to be extracted.

The conditions to be satisfied as defined in chapter 5 are:

- Aliases of objects which are used more than once in a function
- An earlier use of one of the aliases modifies the object
- The alias relation exists at the moment of modification

Now the information needed to determine if the conditions are satisfied has to be collected from the source-code, below the required information for each condition is described.

Aliases of objects which are used more than once in a function

In a function, the use of different variables could lead to changed behavior when the variables are aliases of the same object. To determine if an object is used more than once, a record of all aliases is kept for each variable which is an object. An alias is created when a variable which is an object gets reassigned to another variable.

Besides the aliases, it has to be determined if an object is used more than once. So when is an object used anyway? Each action which can lead to a modified object is seen as 'use of the object'. This means both invocated objects and objects which are used as an argument in a function call qualify for 'use of the object'.

For example calling a method of an object could change some property of the object while an external function could change an object when it is passed as an argument.

For this condition the following facts need to be extracted:

- Aliasing through reassignments (intra-procedural aliasing)
- Function calls (either function or method calls)
- Function call arguments
- Invocated objects

An earlier usage of one of the aliases modified the object

When an object is used multiple times but the object wasn't modified, no changed behavior should occur. To prevent false positives it has to be determined if some use of the object modifies the object, only then changed behavior could occur.

In PHP 4, encapsulation is not enforced. Instead the PHP community has developed naming conventions for private variables and methods. For example the PEAR naming convention [1] dictates private class members are preceded by a single underscore. Why is this important? First take a look at the following definition of encapsulation taken from the glossary of the book Design Patterns [GHJV95] by Gamma et al.

Encapsulation: The result of hiding a representation and implementation in an object. The representation is not visible and cannot be accessed directly from outside the object. Operations are the only way to access and modify an objects representation.

Now besides methods modifying properties of the object they belong to, it is also possible some external function modifies the object. Since programmers are not forced to follow a convention, access restrictions to some private variable could be violated.

Actually in PHP 5 access modifiers have been introduced, methods and properties can be defined as 'public', 'private' or 'protected' but besides the fact PHP 4 source-code needs to be analyzed, it would still be possible for some external function to modify a public property of some object.

So in order to determine what objects are modified, a record is kept of functions which directly modify an object and a record is kept of methods which modify the object they belong to.

Because another function (either a function or class method) can modify some object, the location of the called function is required. A function can be defined in the current file or in a included file so for each call it has to be determined which functions could possibly be called. How this is determined is described in section 6.4, for now only the required facts will be described.

A record is kept for each file and the files which are included by a file. Also the defined functions together with their parameters are stored to determine the inter-procedural aliases at a later stage. Finally the object types are stored which makes it possible to be more accurate which function gets called on a method invocation.

For this condition the following facts need to be extracted:

- Filenames
- Includes
- Functions
- Function parameters (required to perform inter-procedural alias analysis)
- Object types
- Objects which are modified in a function
- Properties a method modifies

The alias relation exists at the moment of modification

Changing an object directly or through a method invocation only causes changed behavior if the alias relation exists at the moment of modification, see figure 5.2 for an example. Therefore a mechanism to

[1] http://pear.php.net/manual/en/standards.naming.php

determine if an alias was already created at the moment of modification is needed. The function calls, variable reassignments and variable modifications are numbered so it is possible to see what was first.

For this condition the following facts need to be extracted:

- Location of function calls and method invocations
- Location of variable reassignments (intra-procedural aliases)
- Location of direct variable modifications

6.3 PHP Fact Extraction

In the first phase facts are extracted from the source-code. A plugin for phc is used to extract the facts from PHP source-files, this process is illustrated in figure 6.2.

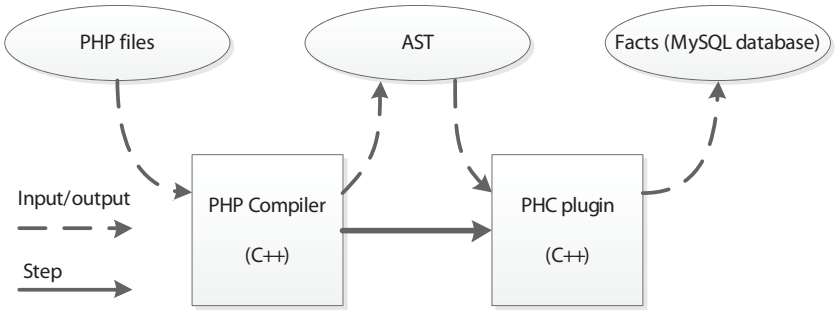


Figure 6.2: Read a PHP program to collect the facts

The facts mentioned in section 6.2 are extracted with the use of a phc plugin. As explained in section 3.2 the plugin uses the Visitor from phc to walk through the AST. Each time the visitor encounters a class definition, a function definition, a function call or an assignment, the plugin will perform the associated actions and store the data in a database.

An important fact about the phc plugin is that it currently only analyzes source-code within functions. Any source-code outside a function is ignored for the time being. This means if there is any source-code defined outside a function which behaves different in PHP 4 compared to PHP 5, the analysis won't find it.

In order to make distinctions between files, function, calls etc. each record gets its unique identifier. References to variables are handled different though. It is likely a variable name is reused in other functions. Therefore the analysis prefixes any reference to a variable with the function id. Meaning when there is some variable 'a' in function 1, the variable will be stored in the database as '1a'.

Any phc plugin has two input parameters, a PHP file and some (optional) string. This means the plugin in is only able to analyze one file in a run and has to be rerun in order to analyze another PHP file. Therefore the unique identifiers are handled by the database because the database knows which identifiers already have been handed out.

Because the phc plugin has to be run separately for each file it probably will be a tremendous task to manually run the plugin for each file in a source-base with hundreds of PHP files. Therefore a script is used to execute the plugin for each file which needs to be analyzed. From a certain directory in the file system all PHP files and the PHP files located in sub directories are read.

6.4 Fact Analysis

In section 6.2 the facts which need to be extracted from PHP source-code are described. This section describes what other facts are needed, why they are needed and what analysis are used to calculate these facts. For an overview of the used analyses please see figure 6.3 in section 6.4.2.

6.4.1 Facts which cannot be retrieved from source-code directly

When some object is used multiple times by its aliases, it has to be determined if the object gets modified somewhere. In section 6.2 we described that the variables modified by a function are stored in the database. Now because passing an object to another function also creates an alias, the object could get modified in the external function.

In order to determine if an object gets modified in an external function, the following information is required:

- Which function gets called at a function call or method invocation
- The objects type (when the call is a method invocation)

Which function gets called at a function call or method invocation

It is not very likely all function names are unique. Different classes could use the same method names and single functions could be defined more than once in different files.

In order to determine which function gets called at a function call or method invocation, a method to determine which functions can be called from a function is needed. In PHP, the parser needs to know where a function is located before it can be used. The function can be defined in the current file or it can be included from another file, for the latter some include statement is used.

The includes will be used to determine to which function definition a function call refers. There is only one difficulty, the includes don't have to be unique. Consider the following include statement:

```
require_once('somefile.php')
```

All files with the name 'somefile.php' now have to be considered. It is possible to prefix the file with a (partial) path but this is not required. The PHP parser solves this by using include paths which means an included file either has to exist in the directory of the file which includes the other file or in one of the include paths.

This analysis wont use the include paths although this could be implemented at a later stage to improve accuracy. Because the include paths can be changed at any place throughout the source-code. It would be necessary to calculate the include paths for every include statement which is quite a tremendous task. So for now the analysis will just include all files with the same name.

There is another aspect about includes which haven't been described, these are the transitive includes. If some file a includes some file b which includes c, c is also accessible from a. When function f calls f2, f2 can exists in the current file or in any of the included files. So the transitive includes also have to be calculated.

The object's type

When a method of some object is invocated, at run-time you know exactly what function is meant because the object's type is available. However, because PHP is dynamic typed, no type information can be retrieved directly from the source-code which makes it difficult to statically determine which method gets called. Below an example is given:

```
function example($obj) {
   $obj->getName();
}

class objectA {
   function getName() {}
}

class objectB {
   function getName() {}
}
```

The function 'example' calls 'getName' on the object 'obj' but there are two classes which both implement 'getName' and because the objects type is unknown, it is impossible to determine which function gets called. In this example both implementations have to be considered, making the analysis less accurate.

When the objects type was known, it is possible to narrow down the possible occurrences of the function 'getName' to classes of the specific type. This could still lead to multiple possibilities because classes can be defined more than once in a source-base. As long as the classes are defined in different files and files which implement the same classes don't get included simultaneously, there is no problem.

Because types are stored by the analysis program, it is possible to calculate an objects type even if the variable is an alias and no direct type information is available. By performing an inter-procedural alias analysis, the possible types of a variable can be calculated. More about this subject is explained in section 6.4.6.

The following additional facts need to be calculated:

- Which includes refers to which file
- Transitive includes
- Which function call refers to which function definition
- Transitive function calls
- What are the inter-procedural aliases

6.4.2 Analyses

To calculate the required facts which cannot be extracted from the source-code directly, different analyses are performed.

Figure 6.3 illustrates what analyses are used, in what order they are performed, what output they generate and which analyses depend on what generated facts. Please note that in order to perform these analyses, additional facts are required. The additional facts are extracted directly from the

source-code and are described in section 6.3 but are left out the illustration to focus on the analyses and their dependencies.

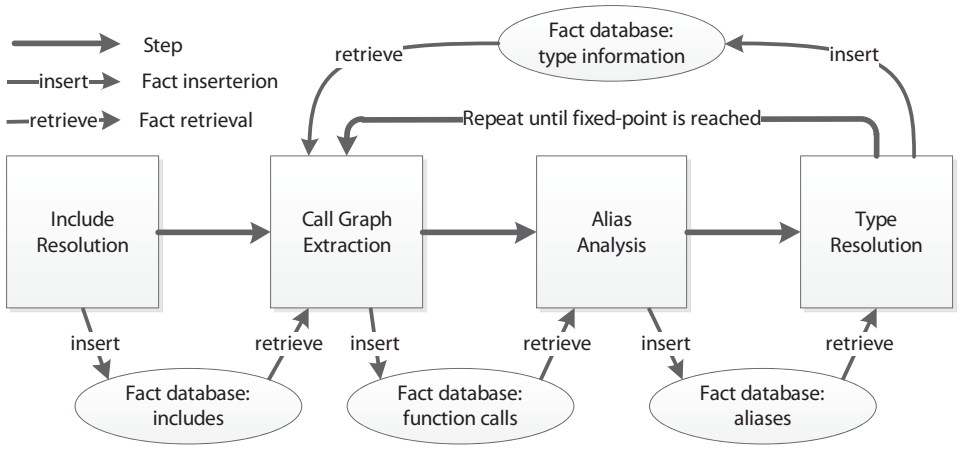


Figure 6.3: Calculate additional facts

Include resolution
Include resolution calculates what files can be reached from a certain file. Include statements in the source-code are linked to file id's and transitive reachable files are calculated. Note that it is possible an include statement is linked to more than one file if the file exists multiple times.

Call graph extraction
The call graph extraction analysis performs two slightly different analyses: function resolution and method resolution. The function resolution analysis determines what function calls refer to what function definitions and uses the include information calculated by the include resolution analysis for more accurate results.

Method resolution determines what method invocations refer to what function definitions and uses both include information calculated by the include resolution analysis and type information which is calculated by the type resolution analysis. When the type of the object - on which the method is invocated - is known, the analysis could determine more precise to which function definition a method invocation refers.

Alias analysis
Alias analysis calculates which variables refer to the same object. Two different alias analyses are performed: intra-procedural alias analysis and inter-procedural alias analysis. Intra-procedural aliases are extracted directly from the source-code, also see section 6.3 but inter-procedural aliases need to be calculated.

Inter-procedural alias analysis uses the function call information generated by the call graph extraction analysis to determine which objects are passed as argument to what functions so it can match the arguments to the function parameters and thus see what aliases are created.

Type analysis
Type analysis calculates the possible object types for a variable. Type information of directly instantiated objects can be extracted from the source-code, see section 6.3 but when an alias variable is created for an object, it is more difficult to determine its type, especially when the alias is created by passing the object as an argument to another function.

This analysis used the alias information generated by the alias analysis to find what aliases a variable has and to determine what aliases are created as direct object instantiations.

Id	Filename	Includes
1	/www/site/a.php	b.php, includes/c.php
2	/www/site/classes/b.php	d.php
3	/www/site/b.php	
4	/www/site/c.php	e.php
5	/www/site/includes/c.php	
6	/www/site/d.php	
7	/www/site/e.php	

Table 6.1: Example of files with includes

Optimizing aliases and method resolution
As described in section 3.3.3, there is a cycle between the alias analysis and the call graph construction analysis which can only be broken by making some initial assumption about the source-code. Figure 6.3 illustrates that first call graph extraction analysis is performed and later type analysis, so during the first run of call graph extraction, no type information is used.

Method resolution analysis might become more accurate because of the type analysis and vice-versa, therefore the analyses are rerun until the function calls stay the same and thus a fixed-point is reached.

The remainder of this chapter covers the different analyses in more detail.

6.4.3 Include resolution

The function definition to which a function call refers to, can exist in the current file or in the included files. When function 'f' calls 'f2', 'f2' can exists in the current file or in any of the included files. Therefore it has to be determined which files are includes by some file.

PHP allows the programmer to specify relative paths to a file and uses include paths to resolve the location of the included files. This analysis does not use the include paths because calculating the possible paths would be a tremendous task since the include paths can be changed at run-time, see section 6.4.1 for greater detail.

In the first phase the includes for each file were extracted, but these includes are just filenames. It is possible some filename exists multiple times, when the include statement doesn't specify the full path to the file it could be ambiguous.

All extracted files are stored with their own unique identifier in the database, as are their includes. Now because the include references are strings, each include statement has to be resolved to a possible file id. In table 6.1 a list of files with the includes for each file is given.

So when is an include statement ambiguous? Consider the include 'b.php', which is included by 'a.php'. In the table there are two files called 'b.php' now which one is needed? This is impossible to say and therefore the analysis has to use them both.

A different example is the other include of 'a.php' which is 'c.php'. This file also exists twice, but because the include statement uses the prefix 'includes/' it is clear which file should be included.

The algorithm to link the include statements to file ids is described in pseudocode below:

```
FOR each file
   FOR each include statement
      FOR each file
         IF include matches file THEN
            store the file id and the matched file id as include in fact database
```

Executing the algorithm on the files from table 6.1 will result in the following ids:

- FILE #1 includes: 2, 3, 5
- FILE #2 includes: 6
- FILE #4 includes: 7

Transitive closure

A called function can exist in the file where the call is being made or in any of the included files. When some file a includes some file b which includes c, c can transitively be reached from a. In order to determine which files can transitively be reached from any given file, the transitive closure has to be calculated.

To calculate the transitive closure, a PHP 5 implementation of the algorithm described in appendix B is used. The analysis takes the includes transformed to file ids as input and then calculates the transitive closure for these files.

Running the program to calculate the transitive closure for the files from table 6.1 results in the following ids:

- FILE #1 includes: 2, 6, 3, 5
- FILE #2 includes: 6
- FILE #4 includes: 7

6.4.4 Call graph extraction

To determine which functions get called, the include information from the PHP source-files is used. The function name has to be unique in a certain context so that it is not ambiguous. This means the function name can only be used once in a single source-file or that it has to be implemented as class method by distinct classes.

To build the call graph for some source-base, a distinction is made between to two types of functions. We call functions which are class members *methods* and we call functions which are not part of a class *functions*. We further refer a call to a function as a *function call* and a call to a method as a *method invocation*.

In order to build the call graph, two types of analyses will be run. A first analysis to determine what functions might be called at some function call, this will be called *function resolution*. A second analysis to determine what functions might be called at some method invocation, this will be called *method resolution*.

Note that while we make a distinction between function resolution and method resolution and also between functions and methods, the definitions of functions and methods are both referred to as *function definitions*.

Naming

Because a function could be defined more than once in some source-base (this is valid as long they are used in a different scope) each function has to receive its own unique identifier. The function name with some additional information will be used as the unique identifier.

As function name, the full path to the file where the function is defined is used. The path is followed by a ':', which is followed by the class name (optional, only when the function is a member of a class thus a method) which is followed by a '+' and finally the actual function name.

When there exists a method 'doSomething()' which is declared in the class 'someClass' in the file 'myfile.php' and myfile.php is located at '/www/site/' the name would be:

/www/site/myfile.php:someClass+doSomething

If on the other hand, the function is defined outside the scope of a class the name would be:

/www/site/myfile.php:+doSomething

Including the class name is needed to distinguish two methods with the same name that are defined in two separate classes but which are located in the same file.

Function resolution

The algorithm to link the function calls to function definition ids is described in pseudocode below:

```
FOR each file
   FOR each function definition in file
      FOR each function call in function definition
         FOR each function definition in included files as callee
            IF function call matches callee THEN
               store the function definition as caller id with callee id in fact
               database
```

To match a function call with a function definition, the analysis takes the defined functions which full names end with the called function name and which are defined in the current or any of the included files which narrows the search space.

Method resolution

Method resolution is basically the same analysis as the analysis performed for function resolution with the addition of using type information. The algorithm to link the method invocations to function definition ids is described in pseudocode below:

```
FOR each file
   FOR each function definition in file
      FOR each method invocation in function definition
         FOR each function definition in included files as callee
            FOR each callee implemented by a class which is a possible type for
            the invocated object
               IF method invocation matches callee THEN
                  store the function definition as caller id with callee id in
                  fact database
```

By using type information, identifying the correct function definition which corresponds to a method invocation becomes more precise because the function definition has to reside in one of the possible classes which narrows the search space.

However, type information is not available when an object is reassigned to another variable or when an object is passed as argument to another function. Only when an object is created and assigned directly assigned to a variable, the variable name and the class name of the instantiated object is stored in the fact database. This information is extracted directly from the source-code, see section 6.2.

In order to determine the possible types of variables which originated from other variables either through reassignments or passing the object as argument to anther function, alias analysis (section section 6.4.5) and type resolution analysis (section 6.4.6) has to be performed. When these analyses have been performed, the types for each variable can be retrieved the fact database.

There is one limitation which makes the analysis less accurate. Aliases are also created when the return value of a function is assigned to a variable and the return value is an object. The analysis currently cannot deal with this way of creating an alias and will register the variable has an unknown type.

So how does this make the analysis less accurate? Well ignoring the unknown type and only using the types which were found could result in false negatives. Because there exists a class which implements the method but is not linked and thus cannot be analyzed.

Instead when a variable has at least one registered unknown type, the type information is ignored and therefore considers methods in all classes instead of only the known classes. The downside of this is more false positives could be introduced and thus results in a less accurate analysis.

Call graph representation

After the function and method resolution analyses are performed, the call graph exists of ids pointing to other ids. Each id represents a function definition where the left id is the calling function/ method and the right ids are the called functions and methods.

It is possible one original function call is now represented by multiple ids when the function and method resolution analyses weren't accurate enough to exactly pinpoint the correct function definition. This could lead to false positives but prevents false negatives.

Transitive closure

To complete the call graph extraction, the transitive closure for the function calls has to be calculated. This is done by the same PHP 5 implementation of the algorithm described in appendix B which is also used to calculate the transitive includes.

Besides an overall transitive call graph, a local call graph is calculated. This so called local call graph only contains methods which were invocated on the object 'this'. No calls between different classes exists, only the internal class internal calls.

The local call graph is needed later to determine if not the invocated method but a transitive reachable method modifies it self. This will be described in more detail in chapter 7.3.2.

6.4.5 Alias analysis

As mentioned previously, alias analysis calculates which variables refer to the same object. The analysis performs two types of alias analyses, intra-procedural alias analysis and inter-procedural alias analysis.

Both analyses use the variable reassignments facts which are extracted by the phc tool. For each variable reassignment the phc tool inserts an alias in two directions, When variable 'b' is an alias for 'a' then 'a' is also an alias for 'b'. This is illustrated in the following code example.

```
$a = new Object();
$b = $a; // Because $b is introduced, both $a and $b point to 'Object'
```

However for type resolution the analysis is less accurate when aliases are stored in both directions, therefore the inverted alias is marked. In the previous example two aliases would be inserted, 'b' pointing to 'a' and 'a' pointing to 'b' but because the latter is inverted, it will be marked.

An explanation on type resolution and why inverted aliases make the analysis less accurate is given in section 6.4.6.

Intra-procedural alias analysis

Intra-procedural alias analysis calculates the aliases created inside a function or method. Actually this is mainly done in the fact extraction phase by storing the variable reassignments (see section 6.2).

Inter-procedural alias analysis

Inter-procedural alias analysis calculates the aliases created when objects are passed as an argument to another function or method. This is done by taking the extracted call graph and matching the function call arguments to the defined function parameters. The algorithm to link function call arguments and method invocation arguments to function definition parameters is described in pseudocode below:

```
FOR each function call
   FOR each function call argument
      IF argument type is object THEN
         IF callee parameter exists with the same index THEN
            store alias pair <parameter, argument> in fact database
```

The function call arguments are stored in a separate table. If the argument is an object, the variable name is stored - and prefixed by the current function id as any reference to a variable - else the base type is stored. A base type could be a integer, float, string, boolean, null or unknown.

During the matching process, only variables which are an object type are matched. Variables with a base type do not have to be linked because these arguments aren't passed by-reference.

Actually two different inter-procedural alias analyses are performed. One which includes the inverted aliases and one which does not include the inverted aliases. Both analysis are stored in separated tables and the latter is used to perform type resolution which is described in section 6.4.6.

Transitive closure

To complete the alias analysis, the transitive aliases have to be calculated. When variable 'b' is an alias for 'a' and variable 'c' is an alias for 'b', 'c' is also an alias for 'a'. This is illustrated in the following code example.

```
$a = new Object();
$b = $a;
$c = $b; // $c is now an alias voor $a and $b
```

As with the transitive includes and the transitive call graph, a PHP 5 implementation of the algorithm described in appendix B is used to calculate the transitive aliases. The transitive closure is calculated for the intra-procedural aliases, the inter-procedural aliases with inversion and the inter-procedural aliases without inversion.

6.4.6 Type resolution

Type resolution analysis is performed to calculate the variable types. When a new object is instantiated, the variable name with its type is stored in the database thus the default variable types are known. But when aliases are created by reassigning variables, the type of alias is unknown. This is illustrated in the following example:

```
$a = new A();
$b = $a; // $b also points to object 'A'
```

Because a new instantiated object is assigned to 'a', the phc plugin sees 'a' has type 'A' which is then stored in the database. But what about 'b'? Since the phc plugin doesn't keep track of aliases, there is no reference from 'b' to the object 'B' only that the variable 'b' is an alias for the variable 'a'. Therefore the alias types will be calculated with a different analysis.

Before the type resolution can be performed, aliases have to be calculated. Not only intra-procedural aliases are needed, but also the inter-procedural aliases are required in order to perform an adequate type analysis. For example take a look at the next example:

```
function f1() {
$a = new A();
   f2($a);
}

function f2($x) {
   $x->getProperty();
}
```

The type of variable 'x' is determined with a parameter of the function 'f2'. In this case 'f2' is called from 'f1' with the argument 'a'. Because the type of variable 'a' is 'A', the possible type for 'x' is also 'A', in order to determine this, inter-procedural alias analysis has to be performed.

However the aliases used to calculate the aliases types should not contain the inverted relations, look at the following two code examples to see why:

```
$a = new Object();
$b = $a;

$a->setName('New name');
```

Calling the method 'setName' on 'a' does not only change 'a' but also 'b' because 'a' and 'b' are both an alias for the same object. Thus not only does 'b' points to 'a', 'a' points to 'b' as well. But to calculate the alias types this cannot be used:

```
$a = new Object();
$b = $a;

$b = new OtherObject();
```

Here 'b' is an alias for 'a' and therefore its type is 'Object' but then a new object is instantiated and assigned to 'b' which has 'OtherObject' as type. Now in the scope of this code 'b' has two types 'Object' and 'OtherObject' but 'a' has only one namely 'Object'. Instantiating the new object 'OtherObject' and assigning it to 'b' breaks the alias causing 'b' to no longer point to the same object as 'a'.

If the inverted relation 'a' pointing to 'b' was used, it would look like 'a' has two types which is incorrect. Therefore the aliases which contain the inverted relations are stored in a separate table. The aliases which contain the inverted relations are used to determine if an object is modified through multiple aliases, this is described in chapter 7.

The algorithm to calculate the possible types for variables is described in pseudocode below:

```
FOR each variable retrieve types of aliases from fact database
   add type information from aliases to variable type information
   store combined type information in fact database
```

The performed analysis is flow insensitive which means that if assigning an object to a variable is based on a condition, the analysis assumes the object always gets assigned. Besides the analysis is context insensitive which means that if different callers pass different object types as an argument to the same function or method parameter, all types are always considered.

6.4.7 Optimizing aliases and method resolution

In section 6.4.4 alias analysis is mentioned as a requirement to perform adequate method resolution analysis. While this is true, in return the method resolution analysis helps improve the alias analysis. So both analyses actually improve each other. How does this work? Take a look at the following code example:

```
function startTeam($leaderName) {
   $team = new Team();
   fillTeam($team, $leaderName);
}

function fillTeam($teamObject, $leader) {
   $person = new Person($leader);
   $teamObject->addTeamMember($person);
}
```

This code is used to create some sort of team and add a default person to it which represents the teamleader. The code is started by calling 'startTeam' with the team leaders name as an argument. Obviously the usefulness of this source-code is debatable, if usefulness was the point. The point here is to clarify how method resolution and alias analysis make both analyses more accurate.

Assume there exists another class called 'SpecialTeam' which also contains the method 'addTeamMember'. Extracting the call graph results in the function call 'fillTeam' being linked to the correct function definition because the name 'fillTeam' is only used once but what about the method invocation 'addTeamMember'?

Because no type information is available yet, the analysis will link the method invocation 'addTeamMember' to the method definition in both 'Team' and 'SpecialTeam'. Note that when the analysis does not know the object type, it will simply link to all classes which contain the method.

After the call graph extraction has been performed, the alias analysis is started. The object 'teamObject' is linked to 'team' while the leader name is ignored because this is a string. Then because the method invocation 'addTeamMember' is linked to the method definitions in both the class 'Team' and 'SpecialTeam', the parameters in both definitions have to be linked to the argument 'person' thus one incorrect alias is added.

Now when rerunning the method resolution analysis, more information is available. The object 'TeamObject' has been linked to the object 'team' in the function 'startTeam'. This objects type is 'Team' therefore the analysis now knows it only has to link the method invocation 'addTeamMember' to the method defined in the class 'Team'.

Rerunning the alias analysis also becomes more accurate because of the improved call graph. Since the method invocation 'addTeamMember' now is linked only to the definition in the class 'Team' no incorrect alias from this method definition in the class 'SpecialTeam' is added.

The new aliases could again improve the analyses thus both the analysis used for call graph extraction and the alias analysis are run continuously until no further improvements occur and a fixed-point is reached.

Chapter 7

Identifying affected code

This chapter describes how the tool identifies source-code which might behave different in PHP 4 compared to PHP 5. Then the achieved results on various test-cases are described and evaluated. This chapter largely answers the main research question:

> **What behavior changes can static source-code analysis precisely identify on a PHP 4 to PHP 5 language migration?**

7.1 Goal

The goal of the analysis is to identify source-code which is subject to the reference change, source-code which actually behaves different in PHP 5 compared to PHP 4 because of the reference change.

Due to the nature of PHP - being a dynamic typed programming language - the analysis sometimes has to make certain conservative assumptions about the source-code, for example to link a method invocation to a method definition. When the possible types of an object contains an unknown type, type information is ignored and all possible classes have to be considered.

The performed analysis is a conservative analysis so that no false negatives exists. This means all source-code which should be identified by the analysis also will be identified. As a side effect, making conservative assumptions will result in more false positives or in other words, source-code which does not behave different in PHP 5 compared to PHP 4 could be incorrectly identified.

While both false positives and false negatives are unwanted and both should be prevented as much as possible, the former is preferred over the latter. The reason false positives are preferred over false negatives is because false positives can be easily checked while false negatives cannot.

When the tool has performed its analysis, the results can be checked. The programmer can look at a piece of source-code which, according to the analysis, behaves different in PHP 5 and validate if the source-code in question actually behaves different. False negatives however cannot be easily checked because because you would have to look through the entire source code and manually check that there is nothing more to identify.

One method to check for false negatives is to manually go over all the source-code and look for source-code which will behave different in PHP 5 and then see whether the code was identified by the analysis. But what would be the point of running the analysis if the source-code still has to be manually checked?

As long as there are not too many false positives it is acceptable to have them. When the analysis con-

tains to many false positives, the analysis becomes unusable because the programmer has to manually check the source-code which could consume a lot of time.

The remainder of this chapter describes how the analysis is performed and evaluates the usefulness of the analysis based on the how reliable the results achieved on the test cases are and how precise the analysis is.

7.2 Basic tool

As proof of concept, a basic tool was build first. The basic tool was used to demonstrate the idea and to gain insight in what source-code it could identify and what not. Knowing what source-code the basic tool could not identify and researching why the tool didn't identify the source-code, helped understand what facts were required to actually do identify that particular source-code.

Below the tool is described and a evaluation is given about the tools performance bases on small test cases.

7.2.1 Method

As described in chapter 6, the phc plugin is used to extract facts from PHP source-code. The basic tool also uses the phc plugin to extract facts from PHP source-code but less facts are extracted. Another difference with the advanced tool is that the phc plugin is not only used to extract facts but also to calculate facts.

The phc plugin makes a profile for each function it encounters, the function name is used as identifier. For each function or method the following data is inserted in a database:

- function name
- boolean property which states if the function/ method modifies it self or one of its parameters
- function call graph
- function call/ method invocation arguments
- invocated object variable at method invocation (target)
- what object variables are used more than once as a call argument or as the target of a method invocation
- type of function call/ method invocation - standard call, argument is reused, target is reused or both argument(s) and target are reused

As can be seen by the extracted facts, the phc tool does more than analyzing PHP source-code and extracting the facts. In contrast to the advanced tool, the basic phc plugin is also used to calculate facts.

For each function, the phc tool keeps track of used objects. When an object is referenced through different aliases, the function call or method invocation gets a special type. This special type means the function call or method invocation uses a variable which might have been changed by a previous call.

When a function call or method invocation does use a variable which might have been changed by a previous call, the variable - a call argument or the target of an method invocation - is stored with the call. This is done to provide information to the user about what variable might have been changed.

The second analysis, performed by a PHP 5 script, uses the data collected by the phc plugin to calculate where possible reference problems might occur. A call is marked as a potential threat when it uses one or more variables multiple times and the calling function has a (transitive) call to at least one function which is marked as a modifying function.

7.2.2 Limitations

The basic tool is limited in the source-code it can identify and especially in its accuracy. Below some causes to the poor performances of the basic tool are described.

Only source-code within functions is analyzed

An important fact about the phc plugin is that it currently only analyzes source-code within functions. Any source-code outside a function is ignored. This means if there is any source-code defined outside a function which behaves different in PHP 4 compared to PHP 5, the analysis won't find it.

However the basic tool isn't alone to suffer from this drawback, the advanced tool has the same limitation.

The function name is used as identifier

Nothing but the function name is used as identifier which means that if a name is used more than once, a distinction between the functions cannot be made. This effects the accuracy of the analysis because when a function name is used more than once and one of the definitions modifies it self or a parameter, all the function definitions are considered to modify them self or a parameter.

Method and function resolution analyses do not narrow the search field

To match a function call or method invocation to a function definition, the analyses does not use include and type information. Since include information is not used, all functions have to be considered when matching a function call to a function definition which making the analysis less accurate.

The method resolution analysis does not use type information which could improve the accuracy of the analysis by using the type information to link a method invocation to a method which is defined in the correct class.

The variables a function modifies are not checked

When an object variable is used through different aliases, changed behavior can only occur when the object is modified. In order to determine if the object could get modified, the analysis checks if the current function or method calls another function which modifies it self or one of its parameters.

Calling a function or method which modifies it self or one of its parameters doesn't necessarily mean the object in question will be modified. A function could modify another parameter or a method could modify it self while the object was passed as an parameter and thus has nothing to do with the changed object.

7.2.3 Small test cases

In order to test how the basic tools performs, the tool is used to analyze small test cases. Each test case represents a different combination of variables which could lead or could not lead to changed behavior. The basis tool is been used to analyze only the first 10 cases, case A till J. The test cases are included in appendix E.

Next the results achieved by the basic tool on the small test cases are discussed, however for an overview of the results please see appendix D.

Summary

The analysis was correct in seven out of ten cases, there were no false negatives but there were three false positives. Three calls were incorrectly identified as a potential reference threat. Mainly due to the previously described limitations of the analysis program.

The analysis program was able to successfully identify reference problems when the modification took place in a transitive called function and when aliasing was used. However some calls were identified as a threat while the variables weren't modified at all.

False identified source-code

The main problem is caused by functions which are thought to modify the object while in fact they do not. The following cases were subject to this particular problem.

- When a new object is created and modified
- When a different parameter is modified (other than the variable that is used more than once)
- When some other function exists with the same name which changes it self or one of its parameters

7.2.4 Evaluation

The problem with the analysis program is that it's not precise enough. It did find all reference threats but also identified function calls as threats while they were no threat at all. The main problem lies in finding if and where a variable gets modified. This is the root problem of the false identified source-code.

The analysis could become more accurate when the call graph extraction would be more precise so there is more certainty a function call is matched to the correct function definition. Besides the conditions on which a function definition is determined to modify an object should be more precise so there is more certainty the changed object is the object in question.

7.3 Advanced tool

In this section the advanced tool is described. First an overview is given of what the tool is supposed to identify and how the analysis tries to achieve this goal. As described in chapter 5, the source-code which needs to be identified has to satisfy the following conditions:

- Aliases of objects which are used more than once in a function

- An earlier use of one of the aliases modifies the object
- The alias relation exists at the moment of modification

In order to identify source-code which satisfies these conditions the analysis uses the extracted facts described in chapter 6. Using the previously extracted facts, the advanced tool is able to locate source-code which satisfies these conditions and output references to the identified source-code which a programmer can use to check if the code really behaves different in PHP 5.

First the analysis identifies objects which are used through different aliases in one function. Then at each usage of the object, the analysis checks if the object could get modified and in the case of the object being the target for a method invocation in the current function, the analysis checks if the alias relation already exist at the moment of modification.

If the object could get modified, the analysis issues a warning saying the previous usage could effect the current usage which could cause changed behavior in PHP 5. The flow of conditions to be satisfied is illustrated in figure 7.1.

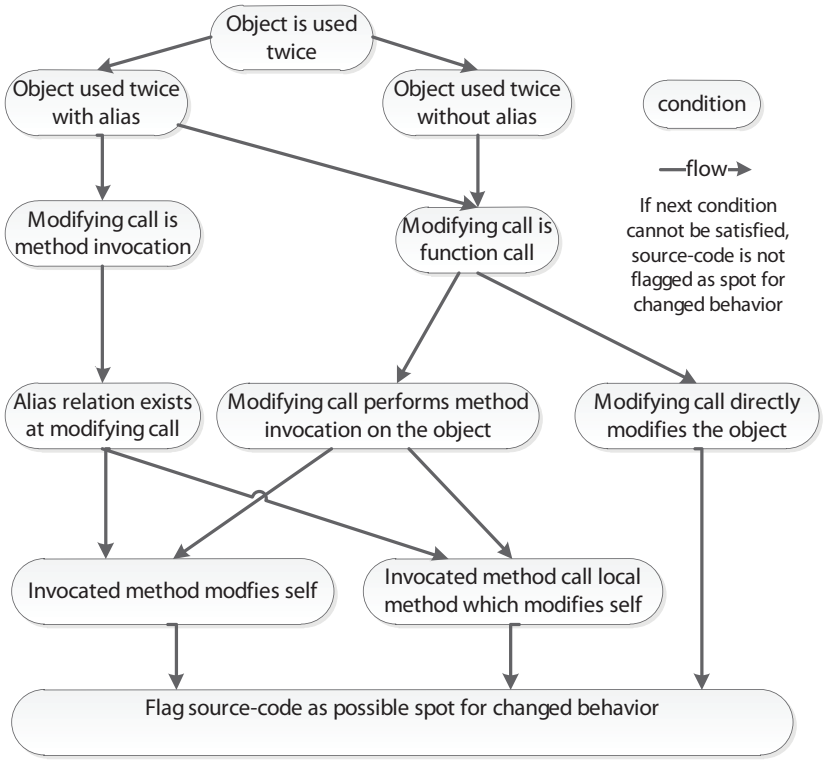


Figure 7.1: Required conditions to identify source-code

7.3.1 Identifying objects which are reused through different aliases

The first analysis performed on the extracted facts is locating the objects which are used through different aliases in the same function, this leads to the next question: *what qualifies as using the object?*

Currently the analysis considers the following actions as using the object:

- The object is used as the target for a method invocation
- The object is used as an argument in a method invocation
- The object is used as an argument in a function call

Method

Below the pseudocode for the algorithm to locate objects which are used multiple times through different aliases is given:

```
FOR each function definition
   FOR each call from function definition

      IF call is method invocation
         retrieve aliases for target
         FOR each alias
            IF alias name exist in used variables list
               store target as reused variable with alias name data
               from used variable list as original used variable

         add target to used variable list


      FOR each argument
         retrieve aliases for argument
         FOR each alias
            IF alias name exist in used variables list
               store argument as reused variable with alias name data
               from used variable list as original used variable

      FOR each argument
         add argument to used variable list
```

When adding the target or argument to the used variable list, the call type (function call or method invocation), the variable name, current call id and statement number are stored. The statement number is used to check if an alias relation exists at the moment of usage and is explained in section 7.3.3.

If an object is used as both target and argument, two usages will be registered. If an object is used multiple times through different arguments in the same call, only one usage will be registered. This is because there is one analysis for checking if a target is modified and one analysis for checking if arguments are modified.

Both analyses have to be performed when the object is used as target and as argument in the same method invocation. But when the same object is used through different arguments, only one usage has to be registered, otherwise this would incorrectly be seen as reusing the object.

Limitations

The list with actions the analysis considers to use the object is not complete. Currently only function calls and method invocations are considered to use the object but directly referencing the object should

also be seen as using the object.

When an object is directly modified by the function, thus without using a setter, changed behavior could occur when the object is used again in the same function. Also when a property of the object is directly requested from the object, thus without using a getter, the object could have changed because of a previous usage.

7.3.2 Identifying which reused objects are modified

When the multiple used objects have been calculated, a different analysis is used to analyze whether a previous usage of some object could modify the object. If it is possible the previous usage modifies the object, the source-code will be flagged as a possible spot of changed behavior.

Mainly there are two analyses which check if an object gets modified, one if the object is the target of a method invocation and one if the object is an argument in a function call or method invocation, the algorithm for the main analysis is given below.

```
FOR each multiple used variable pair
   IF original variable is target of method invocation
      IF original variable is not the reused variable
         IF alias relation exist at original variable location
            IF matched function definitions to method invocation modify themself
               flag source-code

   ELSE
      IF transitive reachable function definitions modify object directly
         flag source-code

      IF transitive reachable function definitions modifies object through
      method invocation
         flag source-code
```

Current function modifies object through setter

When the object is used as the target of some method invocation, the analysis has to determine if the called method modifies it self. To determine this, the analysis requests the variables modified in the called method and looks if any of them references to 'this' meaning a property of the object. If this is true, the source-code will be flagged as a possible spot of changed behavior.

The called method could also call another method of the same object which modifies it self. To check whether the method calls another method which modifies it self, the locally reachable methods are requested from the database and for each of these methods the analysis check if one of them modifies it self. If this is true, the source-code will be flagged as a possible spot of changed behavior.

Other function/ method modifies object directly

When the object is used as an argument of a method invocation or function call, the analysis requests all aliases for the variable and all reachable functions from the current function. With this data, the analysis checks if any of the reachable functions directly modifies one of the aliases.

Directly modifying the alias means a property of an object is directly changed in a function which is not part of the objects class thus without using a setter. If a function does this, the source-code will

be flagged as a possible spot of changed behavior.

Other function/method modifies object through setter

When the object is used as an argument in a method invocation or function call, the object could also be changed with a setter. Now the analysis uses the same data used to see if an object is directly modified. Instead of looking for a reachable function which directly modifies one of the aliases, the analysis now searches for method invocations which exist in the reachable functions and which have one of the aliases as target.

For the found method invocations on the aliases, the analysis checks if these methods modify them self or call another local method which modifies them self, If this is true, the source-code will be flagged as a possible spot of changed behavior.

Limitations

As described in 7.3.1 the analysis currently only considers functions calls and method invocations as using the object, but directly referencing the object should also be seen as using the object.

When an object is directly modified by the function, without using a setter, changed behavior could occur when the object is used again in the same function thus this should be checked. Directly modifying the object is basically the same as using a setter and the source-code should only be identified if the alias relation exist at the moment of modification.

Currently if some object is being modified and reused later, there would be false negatives in the analysis result.

7.3.3 Analyzing if the alias relation exist at moment of modification

When an object is modified directly or with a method invocation and reused later, the alias relation between the variables has to exist at the moment of modification. If the alias relation is established after modifying the object, no changed behavior occurs. See chapter 5.2.3 for a more detailed description.

To determine if the alias relation exists at the moment of modification, the analysis keeps tracks of the order in function calls, method invocations and the creation of aliases. In each function the analysis keeps track of a statement number which is used to number the encountered calls and created aliases.

For each function definition the statement number is reset to 0. Starting with 1, at each function call, method invocation or created alias the phc tool raises the statement number with 1.

The pseudocode for the algorithm to check if the alias relation exists, is given below:

```
retrieve earliest alias pair between reused variable and
original variable or one of it's aliases

IF earliest alias pair location is before original variable location
   alias relation exists before original variable location
```

Now because the statement number is stored with the call which is supposed to modify the object and statement numbers are stored with created aliases, the analysis can check if an alias is created which links the two variables before the function call or method invocation.

If the variables are linked before the method invocation or function call and the call modifies the object, the source-code will be flagged as a possible spot of changed behavior.

Limitations

While the the alias relation has to exist at the moment of modification, the performed analysis is flow insensitive which means that if the creation of the alias is based on a condition, the analysis assumes the alias always gets created.

To prevent false negative, the analysis checks if the alias relation *could* exist at the moment of modification but this conservative method could lead to false positives.

7.3.4 Analysis output

When the analysis is performed, the tool outputs the source-code it identified as possible spot of changed behavior. This includes information about in which file and function definition the behavior might be changed, which function call changes the object and at which call the object is reused.

As described in 7.3.2, the analysis currently detects three types of modification: through a method invocation in the current function or through function call which directly modifies the object or uses a method invocation to modify the object.

Next, the three possible outputs are shown.

FUNCTION CALL WHICH MODIFIED OBJECT THROUGH SETTER

Location:	/testcases/c.php:A+A
Changed behavior site:	At call "getProperty"(60), the object to which variable "84b" points could be changed
Changed behavior cause:	The call "/testcases/c.php:A+DoSomethingWithB"(59) changes object to which variable "84b" points
Matching:	This call is matched to 1 function(s) (details)

Figure 7.2: Identified code: changed in different function with a method invocation

In file "c.php", function "A" the object to which variable "b" points is first used in a different function "DoSomethingWithB" which modifies the object through a setter. Then the object is reused with call "getProperty" thus behavior might have changed.

FUNCTION CALL WHICH DIRECTLY MODIFIES AN OBJECT

Location:	/testcases/b.php:A+A
Changed behavior site:	At call "getProperty"(38), the object to which variable "52b" points could be changed
Changed behavior cause:	The call "/testcases/b.php:A+DoSomethingWithB"(37) changes object to which variable "52b" points
Matching:	This call is matched to 1 function(s) (details)

Figure 7.3: Identified code: changed in different function directly

In file "b.php", function "A" the object to which variable "b" points is first modified in a different function "DoSomethingWithB" which directly changes a property of "b". Then the object is reused with call "getProperty" thus behavior might have changed.

In file "l.php", function "A" the object to which variable "x" points is first modified through an alias "b" with call "setProperty" and then reused with the call "getProperty" thus behavior might have changed.

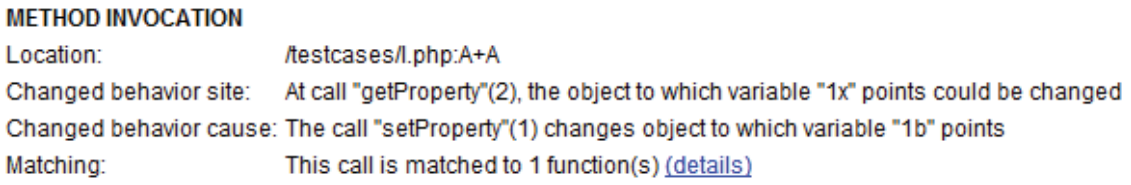


Figure 7.4: Identified code: changed in current function with a method invocation

As can be seen in the examples, the variable names are prefixed with a digit, this is the function definition id to keep the variables unique. Also the function calls are followed by an id, this is the call id. These ids are included for debugging purposes and could be hidden if necessary.

If needed, additional information can be retrieved by clicking on the details link. Because the analysis is conservative, functions calls and method invocations could be matched to more than one function definition. Clicking the details link shows to which function definition a function call is matched and which properties are modified.

A detailed view of an identified spot is given in figure 7.5. It shows the call "setProperty" is matched to a method in the same file which is a member of some class "B" and modifies a property called "_property".

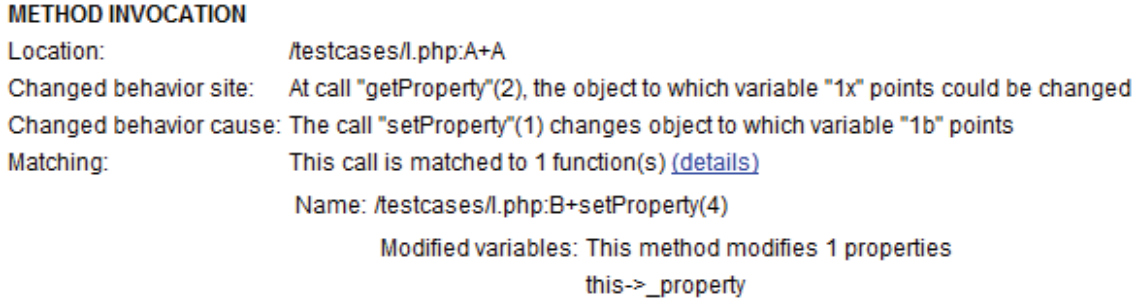


Figure 7.5: Identified code: detailed view

It is also possible a function performs a method invocation which is matched to multiple function definitions. This is illustrated in figure 7.6. Here the method invocation "addItem" is found in two files and it is not clear which file is actually called.

It is up to the programmer to check which file will be called and to determine if changed behavior will occur.

Figure 7.6: Identified code: multiple possible method invocations

Finally it could be possible an object is modified through a method invocation but not by the method itself. If the called method calls another local method which modifies the object, changed behavior

could occur. How this is presented by the analysis tool is shown below.

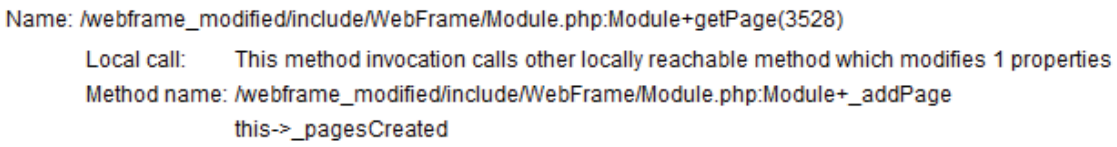


Figure 7.7: Identified code: method invocation call other method which modifies the object

7.3.5 Limitations

Besides the previous described limitations, there are more general limitations which should be mentioned. Modifying an object does not necessarily lead to changed behavior, it leads to a changed object.

If an object is modified but the changed property is not read when reusing the object, no changed behavior would occur but the source-code will be identified as possible spot for changed behavior.

In some cases the aliases could be incomplete. When the alias is created through a return object of some method invocation or function call, the alias is not recognized by the analysis. Also when an alias is created through the loop structure "foreach", the alias is not recognized by the analysis which could lead to false negatives.

Another issue which could lead to false negatives is that inheritance is not recognized by the analysis. Inheritance could cause mismatched function calls if the objects type is known but the method is only defined in the parent class.

This happens because the method resolution analysis uses the class name to match the method invocation to the right class. But when the method definition is not defined in the subclass but in some parent class, the method cannot be matched.

Finally the analysis cannot deal with reflection and will ignore its usage.

7.3.6 Small test cases

In order to test how the advanced tools performs, the tool is used to analyze small test cases. Each test case represents a different combination of variables which could lead or could not lead to changed behavior. The advanced tool analyzes the same test cases used for the basic tool with an additional 12 cases. The test cases are included in appendix E.

Next the results achieved by the advanced tool on the small test cases are discussed, however for an overview of the results please see appendix D.

Summary

The analysis was correct in 17 out of 22 cases. There were 2 false positives and 3 false negatives. The advanced tool is more precise in its analysis than the basic tool, the result is correct for all first 10 test cases while the basic tool contained three false positives on these cases.

Next the causes of the 5 incorrect cases are described.

False positives

First the analysis incorrectly identified 2 spots as code which will behave different in PHP 5 compared to PHP 4 while in fact the behavior is the same.

- case N, modified property is not read
- case V, an array is seen as one object

In case N, an object is modified and read using two variables but the property which is modified at the first call is not read during the second call.

In case V, two objects are stored in one array. One of the objects is modified and one of the objects is read. Because the analysis treats an array as one object, this case is incorrectly identified.

False negatives

- case M, modified property directly in current function
- case Q, alias creation through return object of method invocation
- case T, alias creation through foreach structure

In case M, the object is directly modified in the current function but because only a method invocation or function call is seen as 'using the object', the analysis doesn't recognize the object is used more than once.

In case Q, a getter is used to create two aliases to an object. The analysis cannot deal with aliases created with the returned object of a function. Thus the analysis doesn't recognize the object is used more than once.

In case T, a foreach structure is used to create an alias for a collection of objects but because the analysis cannot deal with this kind of alias creation, it doesn't recognize the object is used more than once.

7.3.7 Real test case

Because the small test cases could be biased by the author of this thesis, the advanced tool is also tested on a larger scale with source-code which is not specially designed as test case but as production code. The achieved results are described in detail in appendix D but below a brief description is given.

The used source-code

The source-code used for the real test-case contains a module from the Clinical Risk Management System developed by The Patient Safety Company and a standard component from their framework. The source-code consists of 42.191 lines of code and is used by multiple health care organizations.

Additional remarks

Because the source-code uses general setup files which include the required files, practically all files are always included. Therefore the include analysis is of no use here and is disabled for performance.

Another issue which must be mentioned is that only function calls and method invocations which have been matched to function definitions can be analyzed. If a function call cannot be matched to a function definition, the analysis cannot tell if the function call would modify the object therefore if the function call is not matched, it is ignored.

So what does this mean for the analysis of the real case? Well it means that only calls to functions which have been analyzed by the tool can be considered thus the functions defined in the module and the functions defined in the component.

Calls inside the component or inside the module will be considered and calls between the module and the component will be considered. But if the module performs a function call to another component, the call is ignored.

The benefit of this approach is that it enables checking only source-code which has been analyzed. This allows the user to analyze only parts of a program if wanted. However this comes at the cost of being less complete.

When only parts of a program are analyzed, it is impossible to say the tool found all spots it is able to find. Thus even if we were 100% sure the tool would find all source-code which would behave different in PHP 5, you still could't say the source-code behaves the same in PHP 4 and PHP 5.

Because the source-code in the real test case are only parts taken from a larger program, it is possible calls are being made to functions which are not part of either the module nor the component thus the analysis ignores these calls.

Finding no possible spots for changed behavior in the real test case doesn't mean there aren't any, it means the tool didn't found these spots where the tool was able to see if the first call modified the object.

Perhaps it would be interesting to show the number of unresolved function calls, this could give a better understanding of how complete the analysis actually is. However such a feature isn't implemented yet.

While we are currently unable to tell how many unresolved function calls exists in the source-code, we can tell how many function calls have been matched. In total there are 9.941 function calls and method invocations being made, these also include calls to native PHP 4 functions though.

The function calls have been matched to 23.088 function definitions which means many function calls have been matched to multiple function definitions.

Summary

The advanced tool found 68 spots which might behave different in PHP 5 compared to PHP 4 but all these spots turned out to be incorrectly identified due to different reasons which will be explained later.

Each spot always consists of a first usage of some object, called the modifying call and a second usage of the object, called the reusing call. At 8 spots the reusing call was actually a native PHP function used to destroy the object, this function is called "unset" and shouldn't cause changed behavior since the object is destroyed instead of read.

Also a modifying call could influence multiple reusing calls and although changed behavior could occur at each spot, the changed behavior can be fixed by only adjusting the modifying call, not the reusing calls. 30 reusing calls shared there modifying call with another reusing call which leaves 30 unique modifying calls.

False positives

Correct identified source-code needs to satisfy four conditions. If one of these conditions is not satisfied, the source-code should not be identified. The conditions are:

- An object is used twice
- A property from the object is modified at the modifying call
- The property which is modified must be read at the reusing call
- The modifying call doesn't pass the object as reference

An object is used twice
The main condition is an object which is used with a modifying call and then reused with a reusing call through a different variable name. At 30 spots this condition was not satisfied for different reasons. Most of these incorrectly identified spots were caused by using an array to store the object.

When different objects are stored in an array, the tool is unable to distinguish the objects in the array and treats them as one. Thus if one object is modified but another is read, no changed behavior would occur.

Another cause for making the tool incorrectly identify an object as being modified and reused is a condition construct. For example an if-then-else structure where either the if-branch or the else-branch is executed. Because the analysis is flow insensitive, it treats all code as if it is always executed.

A property from the object is modified at the modifying call
In 5 out of 38 spots which did use an object twice, no property was modified at the modifying call. The analysis did mark the call as if it modified the object because of multiple reasons.

Matching the function call to function definitions could return multiple function definitions. If the correct function definitions does not modify the object but one of the other matched functions does, the analysis marks the function call as if it modifies the object.

Another reason for incorrectly marking a function call as if it modifies the object is when the variable which points to the object is redefined the function definition matched to the function call. When it is redefined, the analysis treats the variable as if it would still point to the object while in fact it points to a new object.

The property which is modified must be read at the reusing call
In order to get source-code which behaves different in PHP 4 compared to PHP 5, the modified object must be read. Only at 1 of the 38 spots which did use an object twice the modified property was read. So there were 37 spots where the property was not read which means these spots could never lead to changed behavior.

The reason these spots are still identified while they do not read the modified property is because the analysis does not check if the modified property is read. Currently the tool assumes the objects property is read when it was first modified.

The modifying call doesn't pass the object as reference
While the tool searches for objects which have changed due to passing an object by reference instead of by value, this behavior is already possible in PHP 4. Prefixing a parameter of some function definition with an ampersand causes the object being passed by reference instead of PHP 4's default passing by value.

The analysis does not check if an object is already passed by reference thus finding these spots is good is some way but because the current behavior is already as it would be in PHP 5, no changed behavior occurs.

Out of the 38 spots where an object is used twice, only at three spots the object wasn't passed by reference in the modifying call.

False negatives

The existence of false negatives in the real test case has not been tested. Because the used source-code has no automated tests to check whether all code runs the same in PHP 5 as it would in PHP 4, the program would have manually be searched for changed behavior which would take a lot of time and effort.

The small test cases already showed code examples which would not be identified by the tool while they should be identified which makes it likely false negatives also exist in the real test case.

But even if no false negatives would remain in the small test cases, this still wouldn't rule out false negatives in the real test case because it is possible the small test cases are not complete.

However there is one way to do at least some checking for false negatives, which is implementing the small test cases into the real case. There were 11 small test cases where behavior is different and the tool would correctly identify the code so these cases have been implemented in the real test case.

As a result we know where changed behavior should be identified so we could check these spots, if some case would not be identified, a false negative is found. The advanced tool found all implemented spots though.

7.3.8 Evaluation

The goal of the analysis is to identify source-code which is subject to the reference change, source-code which actually behaves different in PHP 5 compared to PHP 4 because of the reference change.

To find source-code which behaves different, a tool was build which should identify all possible spots in the source-code where behavior might have changed. False positives were acceptable as long as there weren't to many of them. However false negatives are not because this means the tool might not have identified all source-code which will behave different in PHP 5.

As for the false positives the results are good. Only 68 spots were identified on a source-base of over 40 000 lines of code. This means less than 1% of the source-code has to be manually checked by a programmer to see if behavior really changes.

And because most of the false positives are caused by either storing the objects in an array or because the objects are already passed by reference, the number of false positives could be reduced even more if the tool would be improved.

As for the false negatives the results are not so good. Because the tool isn't able to identify all spots of changed behavior, running the tool on some source-base and finding no spots for changed behavior doesn't mean there aren't any. However running the analysis does enable the user to fix the spots the tool did identify.

Currently the analysis of the advanced tool isn't sufficient to find all source-code which behaves different in PHP 5. While the presence of false negatives can probably never be completely ruled out, the likelihood of having them is too big.

The current known causes for false negatives are:

- Modifying object property directly in current function is ignored
- Directly reading the property from the object without using a getter is ignored

- Alias creation through return object of method invocation is ignored
- Alias creation through foreach structure ignored
- Inheritance could cause mismatched function calls if object type is know but method is only defined in parent class

Because the source-code which represents these causes is commonly used in the source-base, the likelihood of have false negatives is big. For example using the loop structure "foreach" is often used in the source-code which means aliases could be missing.

However it should be possible to improve the tool so it would be able to deal with these causes for false negatives.

Chapter 8

Summary and Conclusion

The goal of this research was to find out what behavior changes can be identified using static source-code analysis.

Because PHP is a dynamically typed language, it is more difficult to perform name analysis and type analysis. This research shows if the required information to identify source-code which semantics have changed can be retrieved using static source-code analysis.

The main research question to be answered is:

> **What behavior changes can static source-code analysis precisely identify on a PHP 4 to PHP 5 language migration?**

But to answer the main research question, two additional questions have been formulated:

- What changes cause PHP 5 source-code to behave different from PHP 4?
- What facts are needed to identify source-code that behaves different in PHP 5 compared to PHP 4?

The first and second additional research questions are answered in chapter 5 and chapter 6 respectively. While the main research question has largely been answered in chapter 7, a final thought on the main research question is given in the remainder of this chapter.

8.1 Summary

In this project a tool was build which uses static source-code analysis to identify source-code which semantics have changed after PHP language migration. In order to be useful, the analysis result should not contain many false positives and no false negatives. While the number of false positives turned out to be minimal, less than 1% of the analyzed code, false negatives could still remain.

The tool was tested on 22 test cases where each test case represented a unique combination of variables which could or could not lead to changed behavior. The tool did correctly identify 11 cases which would behave different in PHP 5 and correctly skipped 6 cases which would not behave different in PHP 5. However 3 cases were incorrectly skipped by the tool and two cases were incorrectly identified.

Besides small test cases where each case represents a unique combination of variables which could lead to changed behavior, a real test case was used. The source-base consisted of a module from the Clinical Risk Management System developed by The Patient Safety Company and a standard component from their framework.

Because PHP is a dynamic typed programming languages, the tool has to make a lot of conservative assumptions which could lead to false positives. But the result on the real test case showed only 68 spots of source-code which could behave different in PHP 5 compared to PHP 4 which is less than 1% on a source-base of over 40.000 lines of code.

All 68 spots turned out to be false positives caused by various reasons. False negatives have not been tested although we assume they still exist since not all small test cases were identified while they should had. Because the false negatives are caused by such common language structures, it is plausible they also remain in the real test case.

Because both the false negatives and false positives are caused by missing facts, the tools performance could be improved by implementing methods to extract these required facts. Currently running the tool and finding no changed behavior spots doesn't mean there aren't any.

8.2 Conclusion

Before we were able to answer the main research question, we first had to know *"What changes cause PHP 5 source-code to behave different from PHP 4?"*

Chapter 5 showed that the only change which would lead to changed behavior in PHP 5, was that objects are passed and assigned by-reference in PHP 5 instead of by-value in PHP 4.

If an object is passed to a different function which modifies the object, it is also changed at the original site. Using the object again could lead to changed behavior because the object is not the same as it would be in PHP 4.

Then we needed to know *"What facts are needed to identify source-code that behaves different in PHP 5 compared to PHP 4?"* which is described in chapter 6.

Because some false positives and false negatives could be prevented if additional information was available, we can conclude the described required facts are not complete and the tool could be improved if these facts were available. Although this doesn't necessarily mean no false positives or false negatives would remain.

Finally the main research question *"What behavior changes can static source-code analysis precisely identify on a PHP 4 to PHP 5 language migration?"* is largely been answered in chapter 7.

Since only 1 change effected the codes behavior, the main research question only has to be answered for this change. The analysis results shows static source-code analysis can be used to identify source-code which behaves different in PHP 5 because of the pass by reference instead of by-value change.

However currently the analysis is not able to identify all source-code and it comes with the cost of identified source-code which does not behave different in PHP 5. The small test cases show which source-code the tool can identify and which source-code currently cannot be identified.

8.3 Contribution

The research described in this thesis makes several contributions. The main contribution is that we have shown it is possible to use static source-code analysis to identify changed source-code semantics

in the dynamic typed programming language PHP.

The results on the real test case provide insight in how much changed source-code semantics one could expect when migrating a PHP 4 source-base to PHP 5, which is actually less than what we expected.

Finally artifacts of this research are the PHP analysis tools. Also the produced facts can be used for goals other than identifying changed source-code semantics such as dead function detection.

Bibliography

[Age95] O. Agesen. The cartesian product algorithm: Simple and precise type inference of parametric polymorphism. *Proceedings of the 9th European Conference on Object-Oriented Programming*, pages 2 – 26, 1995.

[ASU06] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison Wesley, second edition, 2006.

[Ban79] J. P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Proceedings of the 6th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 29 – 41, 1979.

[Big09] P. Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP.* PhD thesis, Trinity College Dublin, 2009.

[Bin07] D. Binkley. Source code analysis: A road map. *2007 Future of Software Engineering*, pages 104–119, 2007.

[BS96] D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. *ACM SIGPLAN Notices*, 31(10):324 – 341, 1996.

[CBC93] J. D. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 232 – 245, 1993.

[CG93] R. Cytron and R. Gershbein. Efficient accommodation of may-alias information in ssa form. *ACM SIGPLAN Notices*, 28(6):36 – 45, 1993.

[DGC95] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. *Lecture Notes in Computer Science*, 952:77101, 1995.

[dVG07] E. de Vries and J. Gilbert. Design and implementation of a php compiler front-end. Technical report, Trinity College Dublin,, 2007.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 242 – 256, 1994.

[Flo62] R. W. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345, 1962.

[GC01] D. Grove and C. Chambers. A framework for call graph construction algorithms. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(6):685 – 746, 2001.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of reusable object-oriented software.* Addison-Wesley, 1995.

[Han] S. Hanov. Type inference using the cartesian product algorithm on a dynamically typed language.

[Hin01] M. Hind. Pointer analysis: haven't we solved this problem yet? *ACM SIGPLAN - SIGSOFT Workshop on Program Analysis for*, page 5461, 2001.

[HP00] M. Hind and A. Pioli. Which pointer analysis should i use? *Proceedings of the 2000 ACM SIGSOFT international symposium on Software testing and analysis*, pages 113 – 123, 2000.

[JKK06a] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities (short paper). *Proceedings of the 2006 IEEE Symposium on Security and Privacy*, pages 258 – 263, 2006.

[JKK06b] N. Jovanovic, C. Kruegel, and E. Kirda. Precise alias analysis for static detection of web application vulnerabilities. *Proceedings of the 2006 workshop on Programming languages and analysis for security*, pages 27 – 36, 2006.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *ACM SIGPLAN Notices*, 23(7):24 – 31, 1988.

[LR92] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *ACM SIGPLAN Notices*, 27(7):235 – 248, 1992.

[Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[PS91] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. *ACM SIGPLAN Notices*, 26(11):146 – 161, 1991.

[RLS+01] B. G. Ryder, W. A. Landi, P. A. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(2):105 – 186, 2001.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, pages 13 – 22, 1995.

[Ski08] S. Skiena. *The algorithm design manual*. Springer-Verlag, second edition, 2008.

[Sri92] A. Srivastava. Unreachable procedures in object-oriented programming. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1(4):355 – 364, 1992.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(1):1 – 50, 1998.

[Ste96] B. Steensgaard. Points-to analysis in almost linear time. *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 32–41, 1996.

[TLSS99] F. Tip, C. Laffra, P. F. Sweeney, and D. Streeter. Practical experience with an application extractor for java. *ACM SIGPLAN Notices*, 34(10):292 – 305, 1999.

[War62] S Warshall. A theorem on boolean matrices. *Journal of the ACM*, 9(1):11–12, 1962.

[WS07] G. Wassermann and Z. Su. Sound and precise analysis of web applications for injection vulnerabilities. *ACM SIGPLAN Notices*, 42(6):32 – 41, 2007.

Appendices

Appendix A

The Migration Changes

This appendix summarizes most of the changes in PHP5 compared to PHP4. The changes described here have largely been taken from php.net [1] and the web article Whats new in PHP 5 by Andi Gutmans [2]

A. Backward Incompatible Changes

The following changes are divided by being fatal or non-fatal and global or local. Fatal changes will cause to program to halt. Non-fatal changes will throw a warning/ notice and/or might result in unexpected behavior. Global changes have an affect on the whole running the source-code in general and local change only influence certain source-code structures such as a specific function.

Fatal & Global

1. If there are functions defined in the included file, they can be used in the main file independent if they are before return() or after. If the file is included twice, PHP 5 issues fatal error because functions were already declared, while PHP 4 doesn't complain about it. It is recommended to use include_once() instead of checking if the file was already included and conditionally return inside the included file.

Fatal & Local

2. Namespace, abstract, implements, interface, final, public, private, protected, try, catch, throw, instanceof, clone and goto are new reserved keywords.

3. Illegal use of string offsets causes E_ERROR instead of E_WARNING. An example illegal use is: $str = 'abc'; unset(str[0]);.

Non-Fatal & Global

4. Objects are passed by-reference instead of by-value.

[1] http://nl2.php.net/manual/en/migration5.incompatible.php
[2] http://devzone.zend.com/article/1714-Whats-New-in-PHP-5

5. An object with no properties is no longer considered "empty".

6. get_class(), get_parent_class() and get_class_methods() now return the name of the classes and methods as they were declared (casesensitive) which may lead to problems in older scripts that rely on the previous behaviour (the classes and methods name was always returned lowercased). A possible solution is to search for those functions in all your scripts and use strtolower(). This case sensitivity change also applies to the magical predefined constants __CLASS__, __METHOD__, and __FUNCTION__. The values are returned exactly as they're declared (case-sensitive).

7. In some cases classes must be declared before use. It only happens if some of the new features of PHP 5 (such as interfaces) are used. Otherwise the behaviour is the old.

8. include_once() and require_once() first normalize the path of included file on Windows so that including A.php and a.php include the file just once.

Non-Fatal & Local

9. strrpos() and strripos() now use the entire string as a needle.

10. ip2long() now returns FALSE when an invalid IP address is passed as argument to the function, and no longer -1.

11. array_merge() was changed to accept only arrays. If a non-array variable is passed, E_WARNING will be thrown for every such parameter. Be careful because your code may start emitting E_WARNING out of the blue.

12. PATH_TRANSLATED server variable is no longer set implicitly under Apache2 SAPI in contrast to the situation in PHP 4, where it is set to the same value as the SCRIPT_FILENAME server variable when it is not populated by Apache. This change was made to comply with the CGI specification. Please refer to bug #23610 for further information, and see also the $_SERVER ['PATH_TRANSLATED'] description in the manual. This issue also affects PHP versions >= 4.3.2.

13. The T_ML_COMMENT constant is no longer defined by the Tokenizer extension. If error _reporting is set to E_ALL, PHP will generate a notice. Although the T_ML_COMMENT was never used at all, it was defined in PHP 4. In both PHP 4 and PHP 5 // and /* */ are resolved as the T_COMMENT constant. However the PHPDoc style comments /** */, which starting PHP 5 are parsed by PHP, are recognized as T_DOC_COMMENT.

14. $_SERVER should be populated with argc and argv if variables_order includes "S". If you have specifically configured your system to not create $_SERVER, then of course it shouldn't be there. The change was to always make argc and argv available in the CLI version regardless of the variables_order setting. As in, the CLI version will now always populate the global $argc and $argv variables.

B. Backward Compatible Changes

Global

15. **public/private/protected access modifiers for methods and properties**
Allows the use of common OO access modifiers to control access to methods and properties.

```
class MyClass {
    private $id = 18;

    public function getId() {
        return $this->id;
    }
}
```

16. **Unified constructor name __construct()**
    Instead of the constructor being the name of the class, it should now be declared as __construct(), making it easier to shift classes inside class hierarchies.

```
class MyClass {
    function __construct() {
        print "Inside constructor";
    }
}
```

17. **Object destructor support by defining a __destructor() method**
    Allows defining a destructor function that runs when an object is destroyed.

```
class MyClass {
    function __destruct() {
        print "Destroying object";
    }
}
```

18. **Interfaces**
    Gives the ability for a class to fulfill more than one is-a relationships. A class can inherit from one class only but may implement as many interfaces as it wants.

```
interface Display {
    function display();
}

class Circle implements Display {
    function display() {
        print "Displaying circle ";
    }
}
```

19. **final methods**
    The final keyword allows you to mark methods so that an inheriting class can't overload them.

```
class MyClass {
    final function getBaseClassName() {
        return __CLASS__;
    }
}
```

20. **final classes**
    After declaring a class as final, it can't be inherited. The following example would error out:

```
final class FinalClass {
}

class BogusClass extends FinalClass {
}
```

21. **Class constants**
    Classes definitions can now include constant values, and are referenced using the class.

```
class MyClass {
    const SUCCESS = "Success";
    const FAILURE = "Failure";
}
print MyClass::SUCCESS;
```

22. **Static members**
    Classes definitions can now include static members (properties), accessible via the class. Common usage of static members is in the Singleton pattern.

```
class Singleton {
    static private $instance = NULL;

    private function __construct() {
    }

    static public function getInstance() {
        if (self::$instance == NULL) {
            self::$instance = new Singleton();
        }
        return self::$instance;
    }
}
```

23. **Static methods**
    You can now define methods as static allowing them to be called from non-object context. Static methods don't define the $this variable as they aren't bound to any specific object.

```
class MyClass {
    static function helloWorld() {
        print "Hello, world";
    }
}
MyClass::helloWorld();
```

24. **abstract classes**
    A class may be declared as abstract so as to prevent it from being instantiated. However, you may inherit from an abstract class.

```
abstract class MyBaseClass {
    function display() {
        print "Default display routine being called";
    }
}
```

25. **abstract methods**
A method may be declared as abstract, thereby deferring its definition to an inheriting class. A class that includes abstract methods must be declared as abstract.

```
abstract class MyBaseClass {
    abstract function display();
}
```

26. **Class type hints**
Function declarations may include class type hints for their parameters. If the functions are called with an incorrect class type an error occurs.

```
function expectsMyClass(MyClass $obj, String $name) {

}
```

27. **Support for dereferencing objects which are returned from methods**
In PHP 4, you could not directly dereference objects which are returned from methods. You would have to first assign the object to a dummy variable and then dereference it.
PHP 4:

```
$dummy = $obj->method();
$dummy->method2();
```

PHP 5:

```
$obj->method()->method2();
```

28. **Iterators**
PHP 5 allows both PHP classes and PHP extension classes to implement an Iterator interface. Once you implement this interface you will be able to iterate instances of the class by using the foreach() language construct.

```
$obj = new MyIteratorImplementation();
foreach ($obj as $value) {
    print "$value";
}
```

29. **__autoload()**
Many developers writing object-oriented applications create one PHP source file per-class definition. One of the biggest annoyances is having to write a long list of needed includes at the beginning of each script (one for each class). In PHP 5, this is no longer necessary. You may define an __autoload() function which is automatically called in case you are trying to use a class which hasn't been defined yet. By calling this function the scripting engine is giving a last chance to load the class before PHP bails out with an error.

```
function __autoload($class_name) {
    include_once($class_name . "php");
}

$obj  = new MyClass1();
$obj2 = new MyClass2();
```

30. **Exception handling**
PHP 5 adds the ability for the well known try/throw/catch structured exception handling paradigm. You are only allowed to throw objects which inherit from the Exception class.

```
class SQLException extends Exception {
    public $problem;
    function __construct($problem) {
        $this->problem = $problem;
    }
}

try {
    ...
    throw new SQLException("Couldn't connect to database");
    ...
} catch (SQLException $e) {
    print "Caught an SQLException with problem $obj->problem";
} catch (Exception $e) {
    print "Caught unrecognized exception";
}
```

Currently for backwards compatibility purposes most internal functions do not throw exceptions. However, new extensions are making use of this capability and you can use it in your own source code. Also, similar to the already existing set_error_handler() you may use set_exception_handler() to catch an unhandled exception before the script terminates.

31. **foreach with references**
In PHP 4, you could not iterate through an array and modify its values. PHP 5 supports this by allowing you to mark the foreach() loop with the & (reference) sign, thus making any values you change affect the array you're iterating over.

```
foreach ($array as &$value) {
    if ($value === "NULL") {
        $value = NULL;
    }
}
```

32. **default values for by-reference parameters**
In PHP 4, default values could only be given to parameters which are passed by-value. Giving default values to by-reference parameters is now supported.

```
function my_func(&$arg = null) {
    if ($arg === NULL) {
        print '$arg is empty';
    }
}
my_func();
```

Local

33. **instanceof operator**
Language level support for is-a relationship checking. The PHP 4 is_a() function is now deprecated.

```
if ($obj instanceof Circle) {
    print '$obj is a Circle';
}
```

34. **Explicit object cloning**
    In order to clone an object you have to use the clone keyword. You may declare a __clone() method which will be called during the clone process (after the properties have been copied from the original object).

```
class MyClass {
    function __clone() {
        print "Object is being cloned";
    }
}
$obj = new MyClass();
clone $obj;
```

Appendix B

Transitive closure

This appendix describes transitive closure and some algorithms to calculate transitive closure. Also our own implementation of an algorithm to calculate transitive closure is described.

The analyses presented in this thesis represent some of their data in directed graphs. For certain analyses it is necessary to answer some reachability questions about the nodes in the graph, these questions can be answered by calculating the transitive closure for the given graph.

A description of transitive closure is given in [Ski08]:

"Transitive closure can be thought of as establishing a data structure that makes it possible to solve reachability questions (can I get to x from y?) efficiently. After constructing the transitive closure, all reachability queries can be answered in constant time by simply reporting the appropriate matrix entry."

An example graph for calculating the transitive closure to answer certain questions about the graph is a family tree. For instance if someone wants to know whether a person in the tree is the ancestor of some other person.

In mathematics, a relation is transitive if an element 'A' is related to an element 'B' and 'B' is related to an element 'C' then 'A' is also related to 'C'. An example of such a graph is illustrated in figure B.1.

Figure B.1: Transitive relation

Calculating the transitive closure for a given graph will add a direct edge for each pair of nodes which are transitive related. So if 'D' is reachable from 'A' by following one or more nodes, a direct edge between 'A' and 'D' will be added. This process is depicted in figure B.2.

B.1 Algorithms

Next some algorithms to calculate the transitive closure for a directed graph are described.

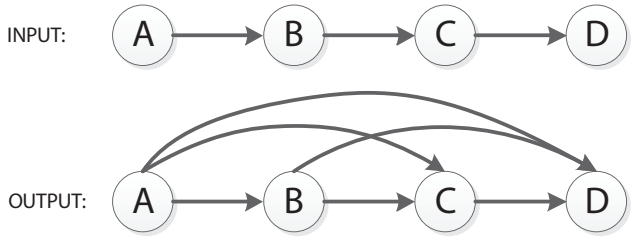
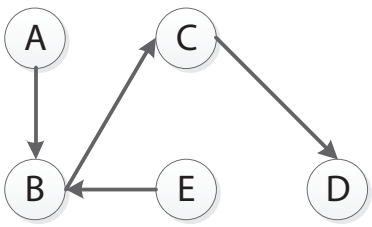


Figure B.2: Constructing transitive closure

B.1.1 Warshall's algorithm

Warshall's algorithm [War62] calculates the transitive closure when given the adjacency matrix of a directed graph (digraph). This algorithm is similar to Floyd's algorithm [Flo62] to calculate the shortest path in a weighted digraph.

Warshall's algorithm calculates the transitive closure for a directed graph which is represented with an adjacency matrix. The matrix shows the relations between the vertices, by writing a 1 for each row-column pair if there exist an edge between them. In figure B.3 a directed graph with it's adjacency matrix is presented.



Directed Graph

	A	B	C	D	E	
A	0	1	0	0	0	AB
B	0	0	1	0	0	BC
C	0	0	0	1	0	CD
D	0	0	0	0	0	-
E	0	1	0	0	0	EB

Adjacency Matrix

Figure B.3: Directed graph and adjacency matrix

To calculate the transitive closure, the algorithm checks for every vertex the incoming and outgoing edges. For each vertex which has both incoming and outgoing edges, a new edge will be added in the matrix by adding a 1 in the corresponding row-column of the new pair.

To illustrate the process an example is given. When the transitive closure for the graph in figure B.3 has to be calculated, the algorithm searches for all vertices with both incoming and outgoing edges.

The first vertex which satisfies this condition is 'B', it has two incoming edges (A and E) and one outgoing edge (C). The algorithm adds two new edges (AC and EC) by adding a 1 in the matrix at the corresponding row-column pair and continues to the next vertex.

The next vertex which has incoming and outgoing edges is 'C', it has three incoming edges (A, B and E) and one outgoing edge (D). See how the incoming edges also include the newly added edges in the previous step. Now the algorithm can add the new edges (AD, BD and ED) and stop because there are no more vertices with both incoming and outgoing edges.

This results in the following matrix:

	A	B	C	D	E
A	0	1	1	1	0
B	0	0	1	1	0
C	0	0	0	1	0
D	0	0	0	0	0
E	0	1	1	1	0

Figure B.4: Adjacency matrix with transitive closure

B.1.2 Depth-First Search

Depth-first search (DFS) is an algorithm to traverse a tree or graph which can be used to calculate the transitive closure. The algorithm to calculate transitive closure performs a depth-first search for each vertex and keeps track of the visited vertices.

The depth-first search algorithm traverses a tree by first visiting the children of each node, starting at the root node. When there are no more children, the algorithm backtracks to the closest not visited node and continues until all nodes are visited.

The figure below illustrates a depth-first traversal through a tree assuming the left children are visited before the right children. The tree traversal order is represented by the numbers in the nodes.

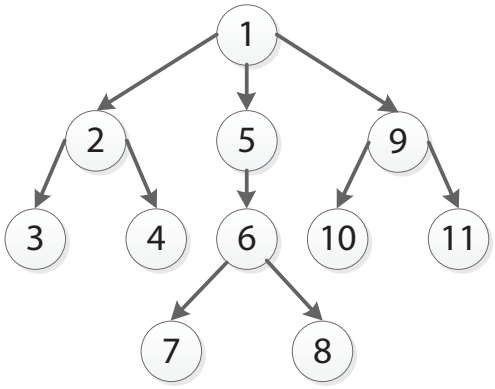


Figure B.5: Depth-first tree traversal

The depth-first search algorithm traverses a graph by following the edges of a given vertex in the graph. When there are no more edges to follow, the algorithm backtracks to the latest unexplored edge and continues there until all edges have been followed.

Look at the graph in figure B.6 where the thick arrows represent edges and the dotted arrows represent the traversal order.

To find the vertices reachable from vertex 'C', the algorithm follows the edges and adds an edge from 'C' to the new vertex. First the algorithm finds 'D' then 'E' then 'F' and finally 'G' thus results in the following edges being added (CD), (CE), (CF) and (CG).

To find the transitive closure for the whole graph, the traversal algorithm has to be performed on each vertex. This has the disadvantage of having to revisit previous visited vertices. For example if the transitive closure has to be calculated for vertex 'D', the algorithm will visit the same vertices as it did when traversing 'C'.

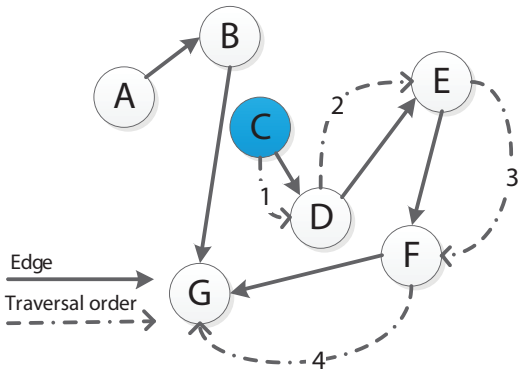


Figure B.6: Depth-first graph traversal

A solution is to remember the reachable vertices from each vertex while traversing the graph. When a vertex has not been visited it can start following the edges else it can reuse the remembered data.

B.2 Algorithm used by the analyses

The algorithm used by the analyses described in this thesis to calculate transitive closure is a really simple but slow. While performance is not the main focus of the analyses described in this thesis, we have build our own implementation of an algorithm to calculate transitive closure.

We believed this algorithm would improve performance, both algorithms are described below:

B.2.1 Simple algorithm

The basic algorithm repeats itself until a fixed-point is reached. When some entity 'A' points to another entity 'B', the algorithm checks to which entities 'B' points to, for example 'X'. Then the algorithm adds an edge between 'A' and 'X'. Now 'A' is related to 'X' and because 'X' could point to other entities, the algorithm is repeated until no further edges are added.

For example if we have some graph with vertices a, b, c and d with the following edges: <a,b>, <b,c> and <c,d>, the edge <a,c> is added in the first iteration of the algorithm but adding the edge <a,d> requires a second iteration of the algorithm.

The algorithm described in pseudo-code:

```
Algorithm buildTransitiveGraphWithComposition(defaultGraph)
   Input: graph
   Output transitive graph

   transitiveGraph = defaultGraph
   tempTransitiveGraph = transitiveGraph
   while tempTransitiveGraph != transitiveGraph
      tempTransitiveGraph = transitiveGraph
      foreach vertex V in transitiveGraph
         foreach vertex C reachable from V
            foreach vertex V' reachable from C
               add edge between V and V' in transitiveGraph
```

```
return transitiveGraph
```

B.2.2 Custom algorithm

Our implementation of the algorithm uses a recursive implementation of the depth-first search algorithm to calculate the transitive closure for each vertex in a graph. While traversing through the graph, the algorithm remembers if a vertex has been visited and stores the reachable vertices from that vertex.

By remembering the reachable vertices from a vertex, the transitive closure for each vertex has to be calculated only once. For example in figure B.6, calculating the transitive closure for vertex 'C' would require the algorithm to visit vertex 'D', 'E', 'F' and 'G' and therefore calculate the transitive closure for these vertices at the same time. By remembering the reachable vertices for each vertex, the vertices don't have to be revisited.

Another reason for remembering visited vertices, is loop detection. Because of the recursive nature of the algorithm, having a cycle in the graph would cause the algorithm to go into an infinite loop. Consider the left image in figure B.7, which depicts a traversal through a graph with a cycle.

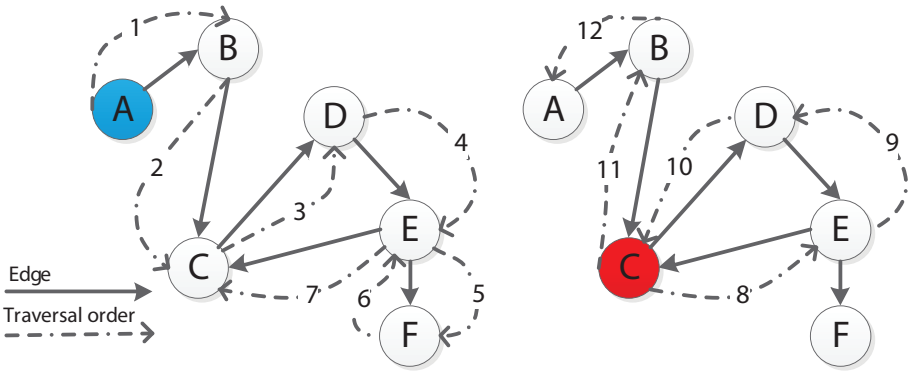


Figure B.7: Graph traversal with cycle

Without loop detection the algorithm would be stuck in the cycle between the vertices 'C', 'D' and 'E'. After revisiting 'C' it would continue to vertex 'D' and therefore never exit. Now because the algorithm keeps track of the previously visited vertices, it can break out the loop by using the visited vertices as a termination condition which is illustrated in the right image in figure B.7.

Note that traversal step 6 doesn't revisit vertex 'E' but simply continues its visit after its first reachable vertex 'F' had no other edges.

At the second visit of vertex 'C', the algorithm detects it has already visited this vertex and therefore the traversal terminates. This causes the algorithm to go back to the previous visited vertex which is 'E'. At 'E' no more vertices have to be visited so it returns to 'D' where no more vertices have to be visited as well and therefore returns to 'C' etc.

However running the algorithm gives an unexpected result. In a cycle all the vertices should be reachable from each other vertex but this is not what happens. Instead any vertex in the graph will only have edges to the vertices visited after the current vertex (this includes the start/end vertex) but not to the earlier visited vertices.

Consider for example the right image in figure B.7. At vertex 'C' the algorithm detects it has already visited this vertex so it terminates the traversal and returns nothing to vertex 'E'. Then 'E' will return

its reachable vertices which are 'C' and 'F' to vertex 'D'. Likewise 'D' returns its reachable vertices which are 'C', 'F' and 'E' to vertex 'C'.

Now the only vertex with the correct edges is 'D' which is easily explained because the only vertex before 'D' is also after 'D', since this is the start/ end vertex of the cycle. Vertex 'E' misses 'D' because this vertex was visited before 'E'. Vertex 'C' is incorrect because it contains itself.

To fix the transitive closure for the vertices in a graph, the algorithm described in B.2.1 is used. For each vertex the recursive algorithm will check if it can reach itself as happened in vertex 'C'. If this is true, the transitive closure for all vertices reachable from the start/ end vertex will be recalculated by composing the default edges and repeat the process until no more edges are added.

The reason why the transitive closure has to be recalculated instead of replacing the reachable nodes from the start/ end vertex with all the other vertices is that there could be vertices reachable from the start/ end vertex which are not in the cycle, for example vertex 'F' in the graph from figure B.7.

The algorithm described in pseudo-code:

```
Algorithm dfs(G)
   Input: graph
   Output transitive graph

   foreach vertex V in G
      if V has not been visited
     transitiveReachableVertices = visit(reachableVerticesFromV)

      if V in transitiveReachableVertices
         recalculateTransitiveClosure(transitiveReachableVertices)
      else
         transitive closure for V = transitiveReachableVertices



visit(vertices, visitedVertices)
   foreach vertex V in vertices
      add V to vertices reachable from parent

      if V has not been visited in the current traversal
         add V to visitedVertices

         if the reachable vertices from V have not been calculated by a previous traversal
            V' = visit(vertices reachable from V, visitedVertices)

            if V in V'
               recalculateTransitiveVertices(V')
            else
               set vertices transitive reachable from V with V'

            add the reachable vertices from V to reachable vertices from parent
         else
            add the reachable vertices from V to reachable vertices from parent
      else
         recalculateTransitiveVertices(visitedVertices)
         reachable vertices from parent is recalculated vertices for V
```

```
   return reachable vertices from parent



recalculateTransitiveVertices(vertices)
   initialize new graph

   foreach vertex V in vertices
      add default edges for V to graph

   transitiveGraph = buildTransitiveGraphWithComposition(graph)

   foreach vertex C in transitiveGraph
      add an transitive reachable vertices from C to transitiveGraph
```

B.2.3 Evaluation

We thought this algorithm would improve performance of our overall analysis because of the recursive nature of the algorithm which would require much less iterations over the nodes than the basic algorithm but testing on large source-bases proofed otherwise. Our algorithm was actually slower than the basic algorithm.

We believe this is caused by the number of cycles in our analyzed data. Because vertices in a cycle have to be fixed by the basic algorithm, the benefit of the recursive algorithm is undone. When the algorithm finds a cycle, the transitive closure for all nodes in the cycle is recalculated.

Appendix C

Database Model

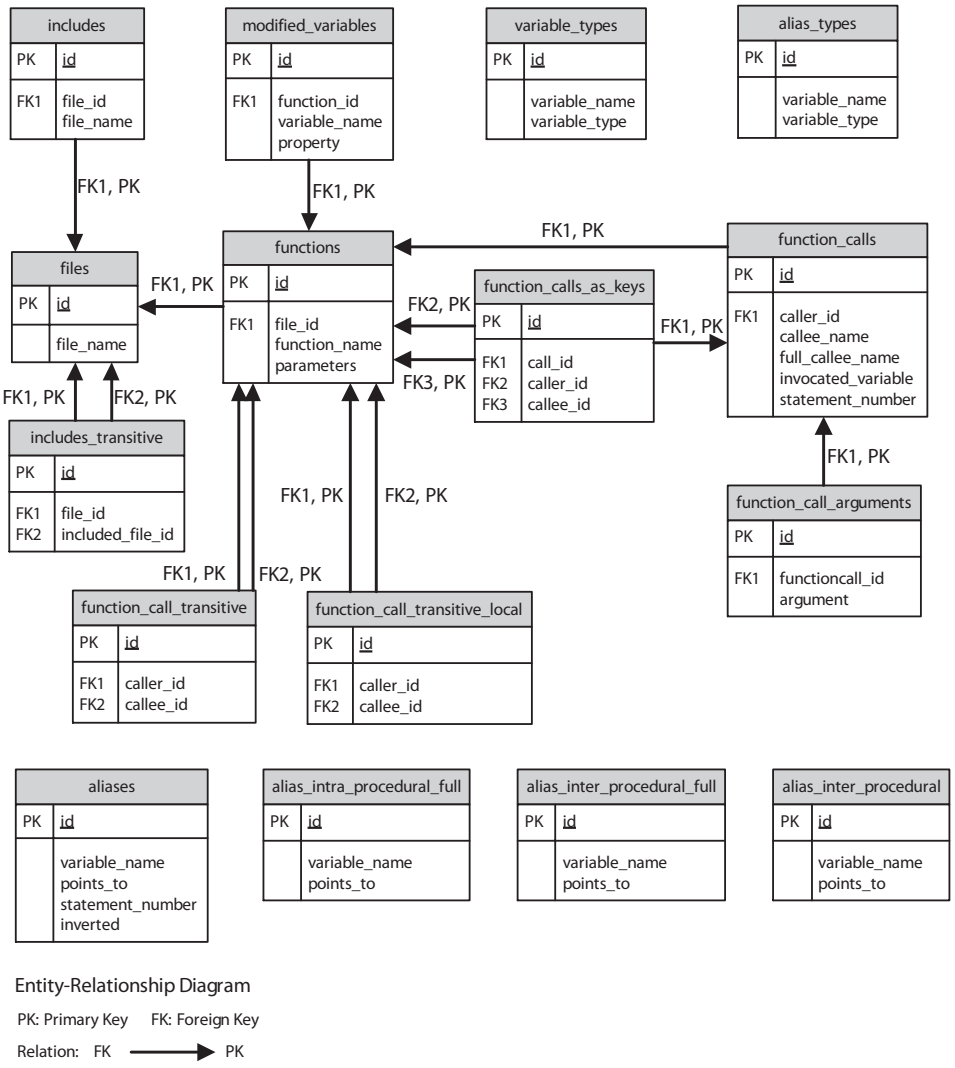


Figure C.1: ERD

Appendix D

Analysis Results

This appendix describes the used small test cases and shows the achieved results on these small test cases and on the real test case.

First an overview is given of the used small test cases. Case A till J have been analyzed by both the basic tool and the advanced tool while the remaining test cases have only been analyzed by the advanced tool.

In table D.1 the small test cases are described by their main properties. The main properties used to describe the different cases are:

Name the name used to describe the case

Changed does the source-code behave different in PHP 5

Object used twice is the same object used twice through different variables

Object modified is the object modified at the modifying call

Object read is the modified object property read

Transitive calls are transitive function calls used

Aliases are aliases used

While these are the main properties, not every combination of properties is made because this would result in cases without additional value. For example the condition that an object has to be used twice through different variables is always required.

Since the analysis only checks the other conditions for which it has determined an object is used twice, there is no point in checking all combinations of properties when the object is not used twice.

As can be seen in table D.1 some cases contain the exact same properties. This is not because they are the same, they are different. The result of a programs can be achieved by different source-code implementations, while the used properties as aliasing for example remain the same.

Consider for example case S and case T. The both contain the same properties but the difference is the used loop structure which modifies the object.

Name	Changed	Object used twice	Object modified	Object read	Transitive	Aliases
A	NO	X	-	-	X	-
B	YES	X	X	X	-	-
C	YES	X	X	X	X	-
D	YES	X	X	X	X	X
E	NO	X	-	-	X	X
F	NO	X	-	-	-	-
G	NO	X	-	-	X	-
H	YES	X	X	X	X	-
I	NO	X	-	-	X	-
J	YES	X	X	X	X	-
K	NO	X	X	-	-	X
L	YES	X	X	X	-	X
M	YES	X	X	X	-	X
N	NO	X	X	-	-	X
O	YES	X	X	X	-	X
P	YES	X	X	X	X	X
Q	YES	X	X	X	-	X
R	YES	X	X	X	-	X
S	YES	X	X	X	X	X
T	YES	X	X	X	X	X
U	YES	X	X	X	X	X
V	NO	-	-	-	X	X

Table D.1: Small test cases

Name	Changed behavior	Identified: basic tool	Correct	Identified: advanced tool	Correct
A	NO	-	X	-	X
B	YES	X	X	X	X
C	YES	X	X	X	X
D	YES	X	X	X	X
E	NO	X	-	-	X
F	NO	X	-	-	X
G	NO	X	-	-	X
H	YES	X	X	X	X
I	NO	-	X	-	X
J	YES	X	X	X	X
K	NO			-	X
L	YES			X	X
M	YES			-	-
N	NO			X	-
O	YES			X	X
P	YES			X	X
Q	YES			-	-
R	YES			X	X
S	YES			X	X
T	YES			-	-
U	YES			X	X
V	NO			X	-
	14/22		7/10		17/22

Table D.2: Results small test-cases

NR	Correct	Used twice	Property modified	Property read	No reference
01	-	X	X	-	-
02	-	X	X	-	-
03	-	X	X	-	-
04	-	X	X	-	-
05	-	X	X	-	-
06	-	X	X	-	-
07	-	X	X	-	-
08	-	X	X	-	-
09	-	X	X	-	-
10	-	X	X	-	-
11	-	X	X	-	-
12	-	X	X	-	-
13	-	X	X	-	-
14	-	X	X	-	-
15	-	X	X	-	-
16	-	-	-	-	-
17	-	-	-	-	-
18	-	-	-	-	-
19	-	-	-	-	-
20	-	-	-	-	-
21	-	X	X	-	-
22	-	X	X	-	-
23	-	X	X	-	-
24	-	X	X	-	-
25	-	X	X	-	-
26	-	X	X	-	-
27	-	X	X	-	-
28	-	X	X	-	-
29	-	-	-	-	-
30	-	-	-	-	-
31	-	-	-	-	-
32	-	-	-	-	-
33	-	-	-	-	-
34	-	X	X	-	-

Table D.3: Results real test-case

NR	Correct	Used twice	Property modified	Property read	No reference
35	-	-	-	-	-
36	-	-	-	-	-
37	-	-	-	-	-
38	-	-	-	-	-
39	-	-	-	-	-
40	-	-	-	-	-
41	-	-	-	-	-
42	-	-	-	-	-
43	-	-	-	-	-
44	-	-	-	-	-
45	-	X	X	-	-
46	-	-	-	-	-
47	-	X	X	-	-
48	-	X	X	-	-
49	-	X	X	-	-
50	-	X	-	-	-
51	-	X	X	-	-
52	-	X	X	-	-
53	-	-	-	-	-
54	-	-	-	-	-
55	-	-	-	-	-
56	-	-	-	-	-
57	-	-	-	-	-
58	-	-	-	-	-
59	-	-	-	-	-
60	-	-	-	-	-
61	-	X	X	-	-
62	-	X	-	-	-
63	-	X	X	-	-
64	-	X	X	-	-
65	-	-	-	-	-
66	-	X	-	X	X
67	-	X	-	-	X
68	-	X	-	-	X
		38x - 56%	33x - 49%	1x - 1.5%	3x - 4.5%

Table D.4: continued results real test-case

Appendix E

Source-code small test cases

In this appendix all used small test cases can be found. For each case first a description is given followed by the source-code. Running a case which has 'Changed behavior' set to 'YES' has a different output when run in PHP 5 compared to PHP 4.

```
<?php
/**
* PHP Analysis - Case A
*
* Changed behavior: NO
*
* Description:
*
* $b is used two times with function calls. "DoSomethingWithB" starts a
* method invocation on the object but since invocation "setProperty" does
* not change the object, no changed behavior occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("newObjectB");

                $this->DoSomethingWithB($b);

                echo $b->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("changedObjectB");
        }
}

class B
{
```

```
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                // not implemented
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case B
*
* Changed behavior: YES
*
* Description:
*
* $b is used two times with function calls. "DoSomethingWithB" directly
* modifies the object which causes changed behavior to occur.
*
*/

class A
{
        function A()
        {
                $b = new B("newObjectB");

                $this->DoSomethingWithB($b);

                echo $b->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->_property = "changedObjectB";
        }
}

class B
{
        var $_property;
```

```
        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                // not implemented
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case C
*
* Changed behavior: YES
*
* Description:
*
* $b is used two times with function calls. "DoSomethingWithB" starts a
* method invocation on the object called "setProperty". Because the
* invocated method changed the object, changed behavior occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $this->DoSomethingWithB($b);

                echo $b->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("ChangedPropertyOfB");
        }
}

class B
{
        var $_property;
```

```
        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case D
*
* Changed behavior: YES
*
* Description:
*
* $b and $c both point to the same object. "DoSomethingWithB" starts a
* method invocation on the object called "setProperty". Because the
* invocated method changed the object, changed behavior occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $this->DoSomethingWithB($b);

                $c = $b;

                echo $c->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("ChangedPropertyOfB");
        }
}

class B
```

```
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case E
*
* Changed behavior: NO
*
* Description:
*
* $b and $c both point to the same object. "DoSomethingWithB" starts a
* method invocation on the object called "setProperty". Because the
* invocated method does not change the object, no changed behavior occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $this->DoSomethingWithB($b);

                $c = $b;

                echo $c->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("ChangedPropertyOfB");
        }
}
```

```
class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $objectVariable = new Something();
                $objectVariable->setVar("Some value");
        }

        function getProperty()
        {
                return $this->_property;
        }
}

class Something
{
        var $_var;

        function Something()
        {
                // not implemented
        }

        function setVar($var)
        {
                $this->_var = $var;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case F
*
* Changed behavior: NO
*
* Description:
*
* Double use of $b in A() but because not $b is modified in
* "DoSomethingWithB" no changed behavior occurs. The variable
* $something actually is modified but this object is not referenced
* more than once.
*
*/
```

```
class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");
                $something = new Something();

                $this->DoSomethingWithB($b, $something);

                echo $b->getProperty();
        }

        function DoSomethingWithB($objectb, $objectSomething)
        {
                $objectSomething->setVar("Some value");
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

class Something
{
        var $_var;

        function Something()
        {
                // do nothing
        }

        function setVar($var)
        {
                $this->_var = $var;
        }
}
```

```
$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case G
*
* Changed behavior: NO
*
* Description:
*
* Double use of $b in A() but calling "setProperty" on an object of
* class B does not change the object, no changed behavior occurs.
* Class C actually has the same method which does modify itself.
*
*/

class A
{
        function A()
        {
                $b = new B();

                $this->DoSomethingWithB($b);

                echo $b->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("ChangedPropertyOfB");
        }
}

class B
{
        var $_property;

        function B()
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                // do nothing
        }

        function getProperty()
        {
                return $this->_property;
        }
}
```

```
class C
{
        var $_property;

        function C()
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case H
*
* Changed behavior: YES
*
* Description:
*
* Double use of $c in A() calling "setProperty" on an object of
* class C does change the object, thus changed behavior occurs.
* Class B actually has the same method which does not modify
* itself.
*
*/

class A
{
        function A()
        {
                $c = new C();

                $this->DoSomethingWithC($c);

                echo $c->getProperty();
        }

        function DoSomethingWithC($objectc)
        {
                $objectc->setProperty("ChangedPropertyOfC");
        }
}
```

```
class B
{
        var $_property;

        function B()
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                // do nothing
        }

        function getProperty()
        {
                return $this->_property;
        }
}

class C
{
        var $_property;

        function C()
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case I
*
* Changed behavior: NO
*
* Description:
*
* Double use of $b in A() where $b is used as parameter twice
* (instead of once as parameter and one method invocation on $b)
```

```
* there are no modifying functions though thus no changed behavior
* occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $this->DoSomethingWithB($b);

                printProperty($b);
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("ChangedPropertyOfB");
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                // not implemented
        }

        function getProperty()
        {
                return $this->_property;
        }
}

function printProperty($object)
{
        echo $object->getProperty();
}

$test = new A();
?>
```

```php
<?php
/**
* PHP Analysis - Case J
*
* Changed behavior: YES
*
* Description:
*
* Double use of $b in A() where $b is used as parameter twice
* (instead of once as parameter and one method invocation on $b)
* and there is a modifying function thus changed behavior does
* occur.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $this->DoSomethingWithB($b);

                printProperty($b);
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->setProperty("ChangedPropertyOfB");
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

function printProperty($object)
```

```
{
        echo $object->getProperty();
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case K
*
* Changed behavior: NO
*
* Description:
*
* Double use of object through alias relation. Because alias relation
* is made after b is modified with the method invocation "setProperty",
* the behavior is the same in PHP 4/5.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $b->setProperty("ChangedPropertyOfB");

                $x = $b;
                echo $x->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->_property = "ChangedPropertyOfB";
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                // not implemented
        }

        function getProperty()
```

```
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case L
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation. Because alias relation
* is made before b is modified with the method invocation "setProperty",
* the behavior is different in PHP 5 compared to PHP 4.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $x = $b;
                $b->setProperty("ChangedPropertyOfB");

                echo $x->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->_property = "ChangedPropertyOfB";
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }
```

```
        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case M
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation. Because alias relation
* is made before b is directly modified by setting the property
* "_property", the behavior is different in PHP 5 compared to PHP 4.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $x = $b;
                $b->_property = "ChangedPropertyOfB";

                echo $x->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->_property = "ChangedPropertyOfB";
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }
```

```
        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case N
*
* Changed behavior: NO
*
* Description:
*
* Double use of object through alias relation. Modifying b could
* influence x but because calling "getProperty" only changes
* "_calledGetProperty" which is not read, no changed behavior
* occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $x = $b;

                echo $b->getProperty();
                echo $x->getProperty();
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->_property = "ChangedPropertyOfB";
        }
}

class B
{
        var $_calledGetProperty = false;
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
```

```
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                $this->_calledGetProperty = true;
                return $this->_property;
        }

        function isCalled()
        {
                return $this->_calledGetProperty;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case O
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation. Calling "getProperty"
* modifies another property than "_property" which is then used to
* determine if the method has already been called. In PHP 5 both x
* and b refer to the same object while in PHP 4 b and x refer to
* different objects, therefore in PHP 5 "$x->isCalled()" will return
* true while in PHP 4 it returns false.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");

                $x = $b;

                if (!$b->isCalled()) {
                        echo $b->getProperty();
                }

                if (!$x->isCalled()) {
                        echo $x->getProperty();
                }
        }

        function DoSomethingWithB($objectb)
```

```
        {
                $objectb->_property = "ChangedPropertyOfB";
        }
}

class B
{
        var $_calledGetProperty = false;
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                $this->_calledGetProperty = true;
                return $this->_property;
        }

        function isCalled()
        {
                return $this->_calledGetProperty;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case P
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation. Calling "getProperty"
* calls another local method which modifies a property which is used
* to determine if the method has already been called. In PHP 5 both x
* and b refer to the same object while in PHP 4 b and x refer to
* different objects, therefore in PHP 5 "$x->isCalled()" will return
* true while in PHP 4 it returns false.
*
*/

class A
{
```

```
        function A()
        {
                $b = new B("SomePropertyOfB");

                $x = $b;

                if (!$b->isCalled()) {
                        echo $b->getProperty();
                }

                if (!$x->isCalled()) {
                        echo $x->getProperty();
                }
        }

        function DoSomethingWithB($objectb)
        {
                $objectb->_property = "ChangedPropertyOfB";
        }
}

class B
{
        var $_calledGetProperty = false;
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                $this->setCalled();
                return $this->_property;
        }

        function isCalled()
        {
                return $this->_calledGetProperty;
        }

        function setCalled()
        {
                $this->_calledGetProperty = true;
        }
}

$test = new A();
```

```
?>
```

```
<?php
/**
* PHP Analysis - Case Q
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation which is assigned by
* the return value of the method "getObject". Because alias relation
* is made before x is modified with the method invocation "setProperty",
* the behavior is different in PHP 5 compared to PHP 4.
*
*/

class A
{
        var $_object;

        function A()
        {
                $this->_object = new B("SomePropertyOfB");

                $x = $this->getObject();
                $y = $this->getObject();

                $x->setProperty("ChangedPropertyOfB");

                echo $y->getProperty();
        }

        function getObject()
        {
                return $this->_object;
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
```

```
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case R
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation which is assigned directly
* from the property "_object" from A. Because alias relation is made
* before x is modified with the method invocation "setProperty", the
* behavior is different in PHP 5 compared to PHP 4.
*
*/

class A
{
        var $_object;

        function A()
        {
                $this->_object = new B("SomePropertyOfB");

                $x = $this->_object;
                $y = $this->_object;

                $x->setProperty("ChangedPropertyOfB");

                echo $y->getProperty();
        }

        function getObject()
        {
                return $this->_object;
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }
```

```
        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case S
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation where one of the variables
* is passed to another function only within an array. Then
* "DoSomethingWithObjects" changes all the objects in the array, thus
* changed behavior occurs.
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");
                $x = $b;
                $objects[] = $b;

                $objects = $this->DoSomethingWithObjects($objects);

                printProperty($x);
        }

        function DoSomethingWithObjects($objects)
        {
                for ($i = 0; $i < count($objects); $i++) {
                        $objects[$i]->setProperty("ChangedPropertyOfB");
                }

                return $objects;
        }
}

class B
{
```

```
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

function printProperty($object)
{
        echo $object->getProperty();
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case T
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation where one of the variables
* is passed to another function only within an array. Then
* "DoSomethingWithObjects" changes all the objects in the array, thus
* changed behavior occurs.
*
* The difference with case S is the kind of loop used in
* "DoSomethingWithObjects".
*
*/

class A
{
        function A()
        {
                $b = new B("SomePropertyOfB");
                $x = $b;
                $objects[] = $b;

                $objects = $this->DoSomethingWithObjects($objects);
```

```
                printProperty($x);
        }

        function DoSomethingWithObjects($objects)
        {
                foreach ($objects as $object) {
                        $object->setProperty("ChangedPropertyOfB");
                }
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

function printProperty($object)
{
        echo $object->getProperty();
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case U
*
* Changed behavior: YES
*
* Description:
*
* Double use of object through alias relation where one of the variables
* is passed to another function only within an array. Then
* "DoSomethingWithObjects" changes one of the objects in the array, this
* is the object which has two variables pointing to it, thus behavior does
* change.
*
*/
```

```
class A
{
        function A()
        {
                $b = new B("B");
                $c = new C("C");

                $x = $b;

                $objects['b'] = $b;
                $objects['c'] = $c;


                $objects = $this->changeObjects($objects);

                printProperty($x);
        }

        function changeObjects($objects)
        {
                $objects['b']->setProperty("ChangedPropertyOfB");

                return $objects;
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

class C
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
```

```
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

function printProperty($object)
{
        echo $object->getProperty();
}

$test = new A();
?>
```

```
<?php
/**
* PHP Analysis - Case V
*
* Changed behavior: NO
*
* Description:
*
* Double use of object through alias relation where one of the variables
* is passed to another function only within an array. Then
* "DoSomethingWithObjects" changes one of the objects in the array, in
* this case this is not the object which has two variables pointing to it,
* thus behavior does not change.
*
*/

class A
{
        function A()
        {
                $b = new B("B");
                $c = new C("C");

                $x = $b;

                $objects['b'] = $b;
                $objects['c'] = $c;


                $objects = $this->changeObjects($objects);

                printProperty($x);
        }
```

```
        function changeObjects($objects)
        {
                $objects['c']->setProperty("ChangedPropertyOfC");

                return $objects;
        }
}

class B
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

class C
{
        var $_property;

        function B($property)
        {
                $this->_property = $property;
        }

        function setProperty($property)
        {
                $this->_property = $property;
        }

        function getProperty()
        {
                return $this->_property;
        }
}

function printProperty($object)
{
        echo $object->getProperty();
}
```

```
$test = new A();
?>
```